TMS320C64x+ DSP Little-Endian DSP Library Programmer's Reference

Literature Number: SPRUEB8
February 2006

TEXAS INSTRUMENTS

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Preface

Read This First

About This Manual

This document describes the C64x+ digital signal processor little-endian (DSP) Library, or DSPLIB for short.

Notational Conventions

This document uses the following conventions:

- ❑ Hexadecimal numbers are shown with the suffix h. For example, the following number is 40 hexadecimal (decimal 64): 40h.
- ❑ Registers in this document are shown in figures and described in tables.
- ❑ Macro names are written in uppercase text; function names are written in lowercase.
  - ■ Each register figure shows a rectangle divded into fields that represent the fields of the register. Each field is labeled with its bit name, its beginning and ending bit numbers above, and its read/write properties below. A legend explains the notation used for the properties.
  - ■ Reserved bits in a register figure designate a bit that is used for future device expansion.

Related Documentation From Texas Instruments

The following books describe the C6000™ devices and related support tools. Copies of these documents are available on the Internet at www.ti.com. *Tip*: Enter the literature number in the search box provided at www.ti.com.

SPRU732 — *TMS320C64x/C64x+ DSP CPU and Instruction Set Reference Guide.* Describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C64x and TMS320C64x+ digital signal processors (DSPs) of the TMS320C6000 DSP family. The C64x/C64x+ DSP generation comprises fixed-point devices in the C6000 DSP platform. The C64x+ DSP is an enhancement of the C64x DSP with added functionality and an expanded instruction set.

SPRAA84 — *TMS320C64x to TMS320C64+ CPU Migration Guide.* Describes migrating from the Texas Instruments TMS320C64x digital signal processor (DSP) to the TMS320C64x+ DSP. The objective of this document is to indicate differences between the two cores. Functionality in the devices that is identical is not included.

Trademarks

Contents

Tables

Chapter 1

Introduction

This chapter provides a brief introduction to the TI C64x+ DSP Libraries (DSPLIB), shows the organization of the routines contained in the library, and lists the features and benefits of the DSPLIB.

Topic **Page**

1.1 Introduction to the TI C64x+ DSPLIB

The TI C64x+ DSPLIB is an optimized DSP Function Library for C programmers using devices that include the C64x+ megamodule. It includes many C-callable, assembly-optimized, general-purpose signal-processing routines. These routines are typically used in computationally intensive real-time applications where optimal execution speed is critical. By using these routines, you can achieve execution speeds considerably faster than equivalent code written in standard ANSI C language. In addition, by providing ready-to-use DSP functions, TI DSPLIB can significantly shorten your DSP application development time.

The TI DSPLIB includes commonly used DSP routines. Source code is provided that allows you to modify functions to match your specific needs.

The routines contained in the library are organized into the following seven different functional categories:

- Adaptive filtering
  - DSP_firlms2
- Correlation
  - DSP_autocor
  - DSP_autocor_rA8
- FFT
  - DSP_fft16x16
  - DSP_fft16x16_imre
  - DSP_fft16x16r
  - DSP_fft16x32
  - DSP_fft32x32
  - DSP_fft32x32s
  - DSP_ifft16x16
  - DSP_ifft16x16_imre
  - DSP_ifft16x32
  - DSP_ifft32x32
  - DSP_fft16x16t (obolete, use DSP_fft16x16)
  - DSP_bitrev_cplx (obsolete, use DSP_fft16x16)
  - DSP_radix 2 (obsolete, use DSP_fft16x16)
  - DSP_r4fft (obsolete, use DSP_fft16x16)
  - DSP_fft (obsolete, use DSP_fft16x16)

- Filtering and convolution
  - DSP_fir_cplx
  - DSP_fir_cplx_hM4X4
  - DSP_fir_gen
  - DSP_fir_gen_hM17_rA8X8
  - DSP_fir_r4
  - DSP_fir_r8
  - DSP_fir_r8_hM16_rM8A8X8
  - DSP_fir_sym
  - DSP_iir
- Math
  - DSP_dotp_sqr
  - DSP_dotprod
  - DSP_maxval
  - DSP_maxidx
  - DSP_minval
  - DSP_mul32
  - DSP_neg32
  - DSP_recip16
  - DSP_vecsumsq
  - DSP_w_vec
- Matrix
  - DSP_mat_mul
  - DSP_mat_trans
- Miscellaneous
  - DSP_bexp
  - DSP_blk_eswap16
  - DSP_blk_eswap32
  - DSP_blk_eswap64
  - DSP_blk_move
  - DSP_fltoq15
  - DSP_minerror
  - DSP_q15tofl

1.2 Features and Benefits

- ❑ Hand-coded assembly-optimized routines
- ❑ C and linear assembly source code
- ❑ C-callable routines, fully compatible with the TI C6x compiler
- ❑ Fractional Q.15-format operands supported on some benchmarks
- ❑ Benchmarks (time and code)
- ❑ Tested against C model

Chapter 2

Installing and Using DSPLIB

This chapter provides information on how to install and rebuild the TI C64x+ DSPLIB.

Topic **Page**

2.1 How to Install DSPLIB

Note:

You should read the README.txt file for specific details of the release.

The DSPLIB is provided in the file dsp64plus.zip. The file must be unzipped to provide the following directory structure:

```
dsp
      |
      +--README.txt      Top-level README file
      |
      +--docs            library documentation
      |
      +--examples        CCS project examples
      |
      |--include         Required include files
      |
      |--lib             library and source archives
      |
      |--support         fft twiddle generation functions
      |
```

Please install the contents of the lib directory in the default directory indicated by your C_DIR environment. If you choose not to install the contents in the default directory, update the C_DIR environment variable, for example, by adding the following line in autoexec.bat file:

```
SET C_DIR=<install_dir>/lib;<install_dir>/include;%C_DIR%
```

or under Unix/csh:

```
setenv C_DIR ”<install_dir>/lib;<install_dir>/include;
$C_DIR”
```

or under Unix/Bourne Shell:

```
C_DIR=”<install_dir>/lib;<install_dir>/include;$C_DIR”;
export C_DIR
```

2.2 Using DSPLIB

2.2.1 DSPLIB Arguments and Data Types

2.2.1.1 DSPLIB Types

Table 2−1 shows the data types handled by the DSPLIB.

Table 2−1. DSPLIB Data Types

Name	Size (bits)	Type	Minimum	Maximum
short	16	integer	−32768	32767
int	32	integer	−2147483648	2147483647
long	40	integer	−549755813888	549755813887
pointer	32	address	0000:0000h	FFFF:FFFFh
Q.15	16	fraction	−0.9999694824...	0.9999694824...
Q.31	32	fraction	−0.99999999953...	0.99999999953...
IEEE float	32	floating point	1.17549435e−38	3.40282347e+38
IEEE double	64	floating point	2.2250738585072014e−308	1.7976931348623157e+308

Unless specifically noted, DSPLIB operates on Q.15-fractional data type elements. Appendix A presents an overview of Fractional Q formats.

2.2.1.2 DSPLIB Arguments

TI DSPLIB functions typically operate over vector operands for greater efficiency. Even though these routines can be used to process short arrays, or even scalars (unless a minimum size requirement is noted), they will be slower for those cases.

- ❑ Vector stride is always equal to 1: Vector operands are composed of vector elements held in consecutive memory locations (vector stride equal to 1).
- ❑ Complex elements are assumed to be stored in consecutive memory locations with Real data followed by Imaginary data.
- ❑ In-place computation is not allowed, unless specifically noted: Source operand cannot be equal to destination operand.

2.2.2 Calling a DSPLIB Function From C

In addition to correctly installing the DSPLIB software, follow these steps to include a DSPLIB function in the code:

- ❑ Include the function header file corresponding to the DSPLIB function
- ❑ Link the code with dsp64plus.lib
- ❑ Use a correct linker command file for the platform used.

The examples in the DSP\Examples folder show how to use the DSPLIB in a Code Composer Studio C envirionment.

2.2.3 Calling a DSP Function From Assembly

The C64x+ DSPLIB functions were written to be used from C. Calling the functions from assembly language source code is possible as long as the calling function conforms to the Texas Instruments C64x+ C compiler calling conventions. For more information, see Section 8 (Runtime Environment) of *TMS320C6000 Optimizing C Compiler User's Guide* (SPRU187).

2.2.4 DSPLIB Testing − Allowable Error

DSPLIB is tested under the Code Composer Studio environment against a reference C implementation. You can expect identical results between Reference C implementation and its Assembly implementation when using test routines that focus on fixed-point type results. The test routines that deal with floating points typically allow an error margin of 0.000001 when comparing the results of reference C code and DSPLIB assembly code.

2.2.5 DSPLIB Overflow and Scaling Issues

The DSPLIB functions implement the same functionality of the reference C code. You must conform to the range requirements specified in the API function, and in addition, restrict the input range so that the outputs do not overflow.

In FFT functions, twiddle factors are generated with a fixed scale factor; i.e., 32767(=2^{15-1}) for all 16-bit FFT functions, 1073741823(=2^{30-1}) for DSP_fft32x32s, 2147483647(=2^{31-1}) for all other 32-bit FFT functions. Twiddle factors cannot be scaled further to not scale input data. Because DSP_fft16x16r and DSP_fft32x32s perform scaling by 2 at each radix-4 stage, the input data must be scaled by $2^{(\log2(nx)-\text{cei}[\log4(nx)-1])}$ to completely prevent overflow. In all other FFT functions, the input data must be scaled by $2^{(\log2(nx))}$ because no scaling is done by the functions.

2.2.6 Interrupt Behavior of DSPLIB Functions

All of the functions in this library are designed to be used in systems with interrupts. Thus, it is not necessary to disable interrupts when calling any of these functions. The functions in the library will disable interrupts as needed to protect the execution of code in tight loops and so on. Library functions have three categories:

- **Fully-interruptible:** These functions do not disable interrupts. Interrupts are blocked by at most 5 to 10 cycles at a time (not counting stalls) by branch delay slots.
- **Partially-interruptible:** These functions disable interrupts for long periods of time, with small windows of interruptibility. Examples include a function with a nested loop, where the inner loop is non-interruptible and the outer loop permits interrupts between executions of the inner loop.
- **Non-interruptible:** These functions disable interrupts for nearly their entire duration. Interrupts may happen for a short time during the setup and exit sequence.

Note that all three function categories tolerate interrupts. That is, an interrupt can occur at any time without affecting the function correctness. The interruptibility of the function only determines how long the kernel might delay the processing of the interrupt.

2.3 How to Rebuild DSPLIB

If you would like to rebuild DSPLIB (for example, because you modified the source file contained in the archive), you will have to use the mk6x utility as follows:

```
mk6x dsp64plus.src –mv64plus –l dsp64plus.lib
```

Chapter 3

DSPLIB Function Tables

This chapter provides tables containing all DSPLIB functions, a brief description of each, and a page reference for more detailed information.

Topic **Page**

3.1 Arguments and Conventions Used

The following convention has been used when describing the arguments for each individual function:

Table 3−1. Argument Conventions

Argument	Description
x,y	Argument reflecting input data vector
r	Argument reflecting output data vector
nx,ny,nr	Arguments reflecting the size of vectors x,y, and r, respectively. For functions in the case nx = ny = nr, only nx has been used across.
h	Argument reflecting filter coefficient vector (filter routines only)
nh	Argument reflecting the size of vector h
w	Argument reflecting FFT coefficient vector (FFT routines only)

Some C64x+ functions have additional restrictions due to optimization using new features such as higher multiply throughput. While these new functions perform better, they can also lead to problems if not carefully used. For example, DSP_autocor_rA8 is faster than DSP_autocor, but the output buffer must be aligned to an 8−byte boundary. Therefore, the new functions are named with any additional restrictions. Three types of restrictions are specified to a pointer: minimum buffer size (M), buffer alignment (A), and the number of elements in the buffer to be a multiple of an integer (X).The following convention has been used when describing the arguments for each individual function:

A kernel function `foo` with two parameters, m and n, with the following restrictions:

m −> Minimum buffer size = 8, buffer alignment = double word, buffer needs to be a multiple of 8 elements

n −> Minimum buffer size = 32, buffer alignment = word , buffer needs to be a multiple of 16 elements

This function would be named: `foo_mM8A8X8_nM32A4X16`.

3.2 DSPLIB Functions

The routines included in the DSP library are organized into eight functional categories and listed below in alphabetical order.

- ❑ Adaptive filtering
- ❑ Correlation
- ❑ FFT
- ❑ Filtering and convolution
- ❑ Math
- ❑ Matrix functions
- ❑ Miscellaneous
- ❑ Obsolete functions

3.3 DSPLIB Function Tables

Table 3−2. Adaptive Filtering

Functions	Description	Page
long DSP_firlms2(short *h, short *x, short b, int nh)	LMS FIR	4-2

Table 3−3. Correlation

Functions	Description	Page
void DSP_autocor(short *r,short *x, int nx, int nr)	Autocorrelation	4-4
void DSP_autocor_rA8(short *r,short *x, int nx, int nr)	Autocorrelation (r[] must be double word aligned)	4-4

Table 3−4. FFT

Functions	Description	Page
void DSP_fft16x16(short *w, int nx, short *x, short *y)	Complex out of place, Forward FFT mixed radix with digit reversal. Input/Output data in Re/Im order.	4-8
void DSP_fft16x16_imre(short *w, int nx, short *x, short *y)	Complex out of place, Forward FFT mixed radix with digit reversal. Input/Output data in Im/Re order.	4-11
void DSP_fft16x16r(int nx, short *x, short *w, unsigned char *brev, short *y, int radix, int offset, int n_max)	Cache-optimized mixed radix FFT with scaling and rounding, digit reversal, out of place. Input and output: 16 bits, Twiddle factor: 16 bits.	4-14
void DSP_fft16x32(short *w, int nx, int *x, int *y)	Extended precision, mixed radix FFT, rounding, digit reversal, out of place. Input and output: 32 bits, Twiddle factor: 16 bits.	4-24
void DSP_fft32x32(int *w, int nx, int *x, int *y)	Extended precision, mixed radix FFT, rounding, digit reversal, out of place. Input and output: 32 bits, Twiddle factor: 32 bits.	4-26
void DSP_fft32x32s(int *w, int nx, int *x, int *y)	Extended precision, mixed radix FFT, digit reversal, out of place., with scaling and rounding. Input and output: 32 bits, Twiddle factor: 32 bits.	4-28

Table 3−4. FFT (Continued)

Functions	Description	Page
void DSP_ifft16x16(short *w, int nx, short *x, short *y)	Complex out of place, Inverse FFT mixed radix with digit reversal. Input/Output data in Re/Im order.	4-28
void DSP_ifft16x16_imre(short *w, int nx, short *x, short *y)	Complex out of place, Inverse FFT mixed radix with digit reversal. Input/Output data in Re/Im order.	4-28
void DSP_ifft16x32(short *w, int nx, int *x, int *y)	Extended precision, mixed radix IFFT, rounding, digit reversal, out of place. Input and output: 32 bits, Twiddle factor: 16 bits.	4-34
void DSP_ifft32x32(int *w, int nx, int *x, int *y)	Extended precision, mixed radix IFFT, digit reversal, out of place, with scaling and rounding. Input and output: 32 bits, Twiddle factor: 32 bits.	4-36

Table 3−5. Filtering and Convolution

Functions	Description	Page
void DSP_fir_cplx (short *x, short *h, short *r, int nh, int nx)	Complex FIR Filter (nh is a multiple of 2)	4-38
void DSP_fir_cplx_hM4X4 (short *x, short *h, short *r, int nh, int nx)	Complex FIR Filter (nh is a multiple of 4)	4-38
void DSP_fir_gen (short *x, short *h, short *r, int nh, int nr)	FIR Filter (any nh)	4-42
void DSP_fir_gen_hM17_rA8X8 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (r[] must be double word aligned, nr must be multiple of 8)	4-42
void DSP_fir_r4 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (nh is a multiple of 4)	4-46
void DSP_fir_r8 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (nh is a multiple of 8)	4-50
void DSP_fir_r8_hM16_rM8A8X8 (short *x, short *h, short *r, int nh, int nr)	FIR Filter (r[] must be double word aligned, nr is a multiple of 8)	4-50
void DSP_fir_sym (short *x, short *h, short *r, int nh, int nr, int s)	Symmetric FIR Filter (nh is a multiple of 8)	4-52

Table 3−5. Filtering and Convolution (Continued)

Functions	Description	Page
void DSP_iir(short *r1, short *x, short *r2, short *h2, short *h1, int nr)	IIR with 5 Coefficients	4-54
void DSP_iirlat(short *x, int nx, short *k, int nk, int *b, short *r)	All−pole IIR Lattice Filter	4-56

Table 3−6. Math

Functions	Description	Page
int DSP_dotp_sqr(int G, short *x, short *y, int *r, int nx)	Vector Dot Product and Square	4-58
int DSP_dotprod(short *x, short *y, int nx)	Vector Dot Product	4-60
short DSP_maxval (short *x, int nx)	Maximum Value of a Vector	4-62
int DSP_maxidx (short *x, int nx)	Index of the Maximum Element of a Vector	4-63
short DSP_minval (short *x, int nx)	Minimum Value of a Vector	4-65
void DSP_mul32(int *x, int *y, int *r, short nx)	32-bit Vector Multiply	4-66
void DSP_neg32(int *x, int *r, short nx)	32-bit Vector Negate	4-68
void DSP_recip16 (short *x, short *rfrac, short *rexp, short nx)	16-bit Reciprocal	4-69
int DSP_vecsumsq (short *x, int nx)	Sum of Squares	4-71
void DSP_w_vec(short *x, short *y, short m, short *r, short nr)	Weighted Vector Sum	4-72

Table 3−7. Matrix

Functions	Description	Page
void DSP_mat_mul(short *x, int r1, int c1, short *y, int c2, short *r, int qs)	Matrix Multiplication	4-73
void DSP_mat_trans(short *x, short rows, short columns, short *r)	Matrix Transpose	4-75

Table 3−8. Miscellaneous

Functions	Description	Page
short DSP_bexp(int *x, short nx)	Max Exponent of a Vector (for scaling)	4-76
void DSP_blk_eswap16(void *x, void *r, int nx)	Endian-swap a block of 16-bit values	4-78
void DSP_blk_eswap32(void *x, void *r, int nx)	Endian-swap a block of 32-bit values	4-80
void DSP_blk_eswap64(void *x, void *r, int nx)	Endian-swap a block of 64-bit values	4-82
void DSP_blk_move(short *x, short *r, int nx)	Move a Block of Memory	4-84
void DSP_fltoq15 (float *x,short *r, short nx)	Float to Q15 Conversion	4-85
int DSP_minerror (short *GSP0_TABLE,short *errCoefs, int *savePtr_ret)	Minimum Energy Error Search	4-87
void DSP_q15tofl (short *x, float *r, short nx)	Q15 to Float Conversion	4-89

Table 3−9. Obsolete Functions

Functions	Description	Page
void DSP_bitrev_cplx (int *x, short *index, int nx)	Use DSP_fft16x16() instead	4-88
void DSP_radix2 (int nx, short *x, short *w)	Use DSP_fft16x16() instead	4-91
void DSP_r4fft (int nx, short *x, short *w)	Use DSP_fft16x16() instead	4-93
void DSP_fft(short *w, int nx, short *x, short *y)	Use DSP_fft16x16() instead	4-96
void DSP_fft16x16t(short *w, int nx, short *x, short *y)	Use DSP_fft16x16() instead	4-107

3.4 Differences Between the C64x and C64x+ DSPLIBs

The C64x+ DSPLIB was developed by optimizing some of the functions of the C64x DSPLIB to take advantage of the C64x+ architecture.

Table 3−10 shows the optimized functions for the C64x+ DSPLIB.

There are two optimization types:

- ❑ SPLOOP conversion: Optimized code uses SPLOOP to provide interruptibility and decrease power consumption. The new C64x+ instructions do not increase algorithm performance, and thus, are not used.
- ❑ Kernel redesign, SPLOOP: Kernel of algorithm rewritten to take advantage of the new C64x+ instructions and of the SPLOOP feature.

Table 3−10. Functions Optimized in the C64x+ DSPLIB

Function	C64x+ Optimized	Optimization Type
DSP_firlms2	No	
DSP_autocor	No	
DSP_autocor_rA8	Yes	Kernel re−design, SPLOOP Optimization resulted in new requirements. New name is used.
DSP_fft16x16	Yes	New Function Optimized C64x+
DSP_fft16x16_imre	Yes	New Function Optimized C64x+
DSP_fft16x16r	Yes	Kernel re−design, SPLOOP
DSP_fft16x32	Yes	Kernel re−design, SPLOOP
DSP_fft32x32	Yes	Kernel re−design, SPLOOP
DSP_fft32x32s	Yes	Kernel re−design, SPLOOP
DSP_ifft16x16	Yes	New Function Optimized C64x+
DSP_ifft16x16_imre	Yes	New Function Optimized C64x+
DSP_ifft16x32	Yes	Kernel re−design, SPLOOP
DSP_ifft32x32	Yes	Kernel re−design, SPLOOP
DSP_fir_cplx	No	

Table 3−10. Functions Optimized in the C64x+ DSPLIB (Continued)

Function	C64x+ Optimized	Optimization Type
DSP_fir_cplx_hM4X4	Yes	Kernel re−design, SPLOOP Optimization resulted in new requirements. New name is used.
DSP_fir_gen	No	
DSP_fir_gen_hM17_rA8X8	Yes	Kernel re−design, SPLOOP Optimization resulted in new requirements. New name is used.
DSP_fir_r4	No	
DSP_fir_r8	No	
DSP_fir_r8_hM16_rM8A8X8	Yes	Kernel re−design, SPLOOP Optimization resulted in new requirements. New name is used.
DSP_fir_sym	No	
DSP_iir	No	
DSP_iirlat	No	
DSP_dotp_sqr	No	
DSP_dotprod	Yes	SPLOOP conversion
DSP_maxval	No	
DSP_maxidx	No	
DSP_minval	No	
DSP_mul32	No	
DSP_neg32	No	
DSP_recip16	No	
DSP_vecsumsq	No	
DSP_w_vec	No	
DSP_mat_mu	No	
DSP_mat_trans	No	
DSP_bexp	No	

Table 3−10. Functions Optimized in the C64x+ DSPLIB (Continued)

Function	C64x+ Optimized	Optimization Type
DSP_blk_eswap16	No	
DSP_blk_eswap32	No	
DSP_blk_move	Yes	SPLOOP conversion
DSP_fltoq15	No	
DSP_minerror	No	
DSP_q15tofl	No	
DSP_bitrev_cplx	No	Obsolete
DSP_radix2	No	Obsolete
DSP_r4fft	No	Obsolete
DSP_fft	No	Obsolete
DSP_fft16x16t	No	Obsolete

Any functions which were not optimized for the C64x+ have the same performance as on the C64x.

Chapter 4

DSPLIB Reference

This chapter provides a list of the functions within the DSP library (DSPLIB) organized into functional categories. The functions within each category are listed in alphabetical order and include arguments, descriptions, algorithms, benchmarks, and special requirements.

Topic **Page**

4.1 Adaptive Filtering

DSP_firlms2 *LMS FIR*

Function long DSP_firlms2(short * restrict h, const short * restrict x, short b, int nh)

Arguments

h[nh]	Coefficient Array
x[nh+1]	Input Array
b	Error from previous FIR
nh	Number of coefficients. Must be multiple of 4.
return long	Return value

Description The Least Mean Square Adaptive Filter computes an update of all nh coefficients by adding the weighted error times the inputs to the original coefficients. The input array includes the last nh inputs followed by a new single sample input. The coefficient array includes nh coefficients.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
long DSP_firlms2(short h[ ],short x[ ], short b,
int nh)
{
    int             i;
    long         r = 0;
    for (i = 0; i < nh; i++) {
        h[i] += (x[i] * b) >> 15;
        r += x[i + 1] * h[i];
    }
    return r;
}
```

Special Requirements

- This routine assumes 16-bit input and output.
- The number of coefficients nh must be a multiple of 4.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- The loop is unrolled 4 times.

Benchmarks

Cycles	3 * nh/4 + 17
Codesize	148 bytes

4.2 Correlation

DSP_autocor *AutoCorrelation*

Function void DSP_autocor(short * restrict r, const short * restrict x, int nx, int nr)

Arguments

r[nr]	Output array
x[nx+nr]	Input array. Must be double-word aligned.
nx	Length of autocorrelation. Must be a multiple of 8.
nr	Number of lags. Must be a multiple of 4.

Description This routine accepts an input array of length nx + nr and performs nr autocorrelations each of length nx producing nr output results. This is typically used in VSELP code.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_autocor(short r[ ],short x[ ], int nx, int nr)
{
    int i,k,sum;
    for (i = 0; i < nr; i++){
        sum = 0;
        for (k = nr; k < nx+nr; k++)
            sum += x[k] * x[k-i];
        r[i] = (sum >> 15);
    }
}
```

Special Requirements

- nx must be a multiple of 8.
- nr must be a multiple of 4.
- x[ ] must be double-word aligned.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- The inner loop is unrolled 8 times.
- The outer loop is unrolled 4 times.
- The outer loop is conditionally executed in parallel with the inner loop. This allows for a zero overhead outer loop.

Benchmarks

Cycles	nx<40:	6*nr*nr/4 + 20
	nx>=40:	nx*nr/8 + 2*nr + 20
Codesize	304 bytes	

DSP_autocor_rA8 *AutoCorrelation*

Function void DSP_autocor_rA8(short * restrict r, const short * restrict x, int nx, int nr)

Arguments

r[nr]	Output array, Must be double word aligned.
x[nx+nr]	Input array. Must be double-word aligned.
nx	Length of autocorrelation. Must be a multiple of 8.
nr	Number of lags. Must be a multiple of 4.

Description This routine accepts an input array of length nx + nr and performs nr autocorrelations each of length nx producing nr output results. This is typically used in VSELP code.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_autocor(short r[ ],short x[ ], int nx, int nr)
{
    int i,k,sum;
    for (i = 0; i < nr; i++){
        sum = 0;
        for (k = nr; k < nx+nr; k++)
            sum += x[k] * x[k-i];
        r[i] = (sum >> 15);
    }
}
```

Special Requirements

- ❑ nx must be a multiple of 8.
- ❑ nr must be a multiple of 4.
- ❑ x[ ] must be double-word aligned.
- ❑ r[ ] must be double-word aligned.

Implementation Notes

- ❑ ***Bank Conflicts:*** No bank conflicts occur.
- ❑ ***Interruptibility:*** The code is interruptible.
- ❑ The inner loop is unrolled 8 times.
- ❑ The outer loop is unrolled 4 times.

Benchmarks	Cycles	nx<40: 6*nr+ 20
		nx>=40: nx*nr/8 + 2*nr + 20
	Codesize	304 bytes

4.3 FFT

DSP_fft16x16 *Complex Forward Mixed Radix 16 x 16-bit FFT*

Function void DSP_fft16x16(const short * restrict w, int nx, short * restrict x, short * restrict y)

Arguments

w[2*nx]	Pointer to complex Q.15 FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4 , and 16 ≤ nx ≤ 32768.
x[2*nx]	Pointer to complex 16-bit data input.
y[2*nx]	Pointer to complex 16-bit data output.

Description This routine computes a complex forward mixed radix FFT with rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache.

Algorithm All stages are radix-4 except the last one, which can be radix-2 or radix-4, depending on the size of the FFT. All stages except the last one scale by two the stage output data.

Special Requirements

- In-place computation is *not* allowed.
- The size of the FFT, nx, must be power of 2 or 4, and 16 ≤ nx ≤ 32768.
- The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices. All data are in short precision or Q.15 format.

Implementation Notes

- ❏ ***Bank Conflicts:*** No bank conflicts occur.
- ❏ ***Interruptibility:*** The code is interruptible.

The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform. The conventional Cooley Tukey FFT is written using three loops. The outermost loop “k” cycles through the stages. There are log N to the base 4 stages in all. The loop “j” cycles through the groups of butterflies with different twiddle factors, and loop “i” reuses the twiddle factors for the different butterflies within a stage. Note the following:

Stage	Groups	Butterflies With Common Twiddle Factors	Groups*Butterflies
1	N/4	1	N/4
2	N/16	4	N/4
..	..	..	..
logN	1	N/4	N/4

The following statements can be made based on above observations:

1) Inner loop “i0” iterates a variable number of times. In particular, the number of iterations quadruples every time from 1..N/4. Hence, software pipelining a loop that iterates a variable number of times is not profitable.

2) Outer loop “j” iterates a variable number of times as well. However, the number of iterations is quartered every time from N/4 ..1. Hence, the behavior in (a) and (b) are exactly opposite to each other.

3) If the two loops “i” and “j” are coalesced together then they will iterate for a fixed number of times, namely N/4. This allows us to combine the “i” and “j” loops into one loop. Optimized implementations will make use of this fact.

In addition,, the Cooley Tukey FFT accesses three twiddle factors per iteration of the inner loop, as the butterflies that reuse twiddle factors are lumped together. This leads to accessing the twiddle factor array at three points, each separated by “ie”. Note that “ie” is initially 1, and is quadrupled with every iteration. Therefore, these three twiddle factors are not even contiguous in the array.

To vectorize the FFT, it is desirable to access the twiddle factor array using double word wide loads and fetch the twiddle factors needed. To do this, a modified twiddle factor array is created, in which the factors WN/4, WN/2, W3N/4 are arranged to be contiguous. This eliminates the separation between twiddle factors within a butterfly. However, this implies that we maintain a redundant version of the twiddle factor array as the loop is traversed from one stage to another. Hence, the size of the twiddle factor array increases as compared to the normal Cooley Tukey FFT. The modified twiddle factor array is of size "2 * N" where the conventional Cooley Tukey FFT is of size "3N/4" where N is the number of complex points to be transformed. The routine that generates the modified twiddle factor array was presented earlier. With the above transformation of the FFT, both the input data and the twiddle factor array can be accessed using double-word wide loads to enable packed data processing.

The final stage is optimized to remove the multiplication as w0 = 1. This stage also performs digit reversal on the data, so the final output is in natural order. In addition, if the number of points to be transformed is a power of 2, the final stage applies a radix-2 pass instead of a radix-4. In any case, the outputs are returned in normal order.

The code performs the bulk of the computation in place. However, because digit-reversal cannot be performed in-place, the final result is written to a separate array, y[].

Benchmarks

Cycles	$(6 * nx/8 + 19) * \text{ceil}[\log_4(nx) - 1] + 8*nx/8 + 30$
Codesize	864 bytes

DSP_fft16x16_imre *Complex Forward Mixed Radix 16 x 16-bit FFT, With Im/Re Order*

Function void DSP_fft16x16_imre(const short * restrict w, int nx, short * restrict x, short * restrict y)

Arguments

w[2*nx]	Pointer to complex Q.15 FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4 , and 16 ≤ nx ≤ 32768.
x[2*nx]	Pointer to complex 16-bit data input.
y[2*nx]	Pointer to complex 16-bit data output.

Description This routine computes a complex forward mixed radix FFT with truncation and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved **imaginary** and **real** parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache.

Algorithm All stages are radix-4 except the last one, which can be radix-2 or radix-4, depending on the size of the FFT. All stages except the last one scale by two the stage output data.

Special Requirements

- In-place computation is *not* allowed.
- The size of the FFT, nx, must be power of 2 or 4, and 16 ≤ nx ≤ 32768.
- The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- The input and output data are complex, with the **imaginary/real** components stored in adjacent locations in the array. The imaginary components are stored at even array indices, and the real components are stored at odd array indices. All data are in short precision or Q.15 format.

Implementation Notes

- ***Bank Conflicts:*** no conflicts occur.
- ***Interruptibility:*** The code is interruptible.

The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform. The conventional Cooley Tukey FFT is written using three loops. The outermost loop “k” cycles through the stages. There are log N to the base 4 stages in all. The loop “j” cycles through the groups of butterflies with different twiddle factors, and loop “i” reuses the twiddle factors for the different butterflies within a stage. Note the following:

Stage	Groups	Butterflies With Common Twiddle Factors	Groups*Butterflies
1	N/4	1	N/4
2	N/16	4	N/4
..	..	..	..
logN	1	N/4	N/4

The following statements can be made based on above observations:

1) Inner loop “i0” iterates a variable number of times. In particular, the number of iterations quadruples every time from 1..N/4. Hence, software pipelining a loop that iterates a variable number of times is not profitable.

2) Outer loop “j” iterates a variable number of times as well. However, the number of iterations is quartered every time from N/4 ..1. Hence, the behavior in (a) and (b) are exactly opposite to each other.

3) If the two loops “i” and “j” are coalesced together then they will iterate for a fixed number of times, namely N/4. This allows us to combine the “i” and “j” loops into one loop. Optimized implementations will make use of this fact.

In addition, the Cooley Tukey FFT accesses three twiddle factors per iteration of the inner loop, as the butterflies that reuse twiddle factors are lumped together. This leads to accessing the twiddle factor array at three points, each separated by “ie”. Note that “ie” is initially 1, and is quadrupled with every iteration. Therefore these three twiddle factors are not even contiguous in the array.

To vectorize the FFT, it is desirable to access twiddle factor array using double word wide loads and fetch the twiddle factors needed. To do this, a modified twiddle factor array is created, in which the factors WN/4, WN/2, W3N/4 are arranged to be contiguous. This eliminates the separation between twiddle factors within a butterfly. However, this implies that we maintain a redundant version of the twiddle factor array as the loop is traversed from one stage to another. Hence, the size of the twiddle factor array increases as compared to the normal Cooley Tukey FFT. The modified twiddle factor array is of size “2 * N”, where the conventional Cooley Tukey FFT is of size “3N/4”, where N is the number of complex points to be transformed. The routine that generates the modified twiddle factor array was presented earlier. With the above transformation of the FFT, both the input data and the twiddle factor array can be accessed using double-word wide loads to enable packed data processing.

The final stage is optimized to remove the multiplication as w0 = 1. This stage also performs digit reversal on the data, so the final output is in natural order. In addition, if the number of points to be transformed is a power of 2, the final stage applies a DSP_radix2 pass instead of a radix 4. In any case, the outputs are returned in normal order.

The code performs the bulk of the computation in place. However, because digit-reversal cannot be performed in-place, the final result is written to a separate array, y[].

Benchmarks

Cycles	(6 * nx/8 + 19) * ceil[$\log_4$(nx) − 1] + 8*nx/8 + 30
Codesize	864 bytes

DSP_fft16x16r *Complex Forward Mixed Radix 16 x 16-bit FFT With Rounding*

Function void DSP_fft16x16r(int nx, short * restrict x, const short * restrict w, const unsigned char * restrict brev, short * restrict y, int radix, int offset, int nmax)

Arguments

nx	Length of FFT in complex samples. Must be power of 2 or 4, and ≤16384
x[2*nx]	Pointer to complex 16-bit data input
w[2*nx]	Pointer to complex FFT coefficients
brev[64]	Pointer to bit reverse table containing 64 entries. Only required for C code. Use NULL for assembly code since BITR instruction is used instead.
y[2*nx]	Pointer to complex 16-bit data output
radix	Smallest FFT butterfly used in computation used for decomposing FFT into sub-FFTs. See notes.
offset	Index in complex samples of sub-FFT from start of main FFT.
nmax	Size of main FFT in complex samples.

Description This routine implements a cache-optimized complex forward mixed radix FFT with scaling, rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored as interleaved 16-bit real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors).

This redundant set of twiddle factors is size 2*N short samples. As pointed out in subsequent sections, dividing these twiddle factors by 2 will give an effective divide by 4 at each stage to guarantee no overflow. The function is accurate to about 68dB of signal to noise ratio to the DFT function as follows.

```
void dft(int n, short x[], short y[])
{
   int k,i, index;
   const double PI = 3.14159654;
   short * p_x;
   double arg, fx_0, fx_1, fy_0, fy_1, co, si;

   for(k = 0; k<n; k++)
   {
     p_x = x;
     fy_0 = 0;
     fy_1 = 0;
     for(i=0; i<n; i++)
     {
       fx_0 = (double)p_x[0];
       fx_1 = (double)p_x[1];
       p_x += 2;
       index = (i*k) % n;
       arg = 2*PI*index/n;
       co = cos(arg);
       si = -sin(arg);
       fy_0 += ((fx_0 * co) - (fx_1 * si));
       fy_1 += ((fx_1 * co) + (fx_0 * si));
     }
     y[2*k] = (short)2*fy_0/sqrt(n);
     y[2*k+1] = (short)2*fy_1/sqrt(n);
   }
}
```

Scaling by 2 (i.e., >>1) takes place at each radix-4 stage except the last one. A radix-4 stage could give a maximum bit-growth of 2 bits, which would require scaling by 4. To completely prevent overflow, the input data must be scaled by $2^{(BT-BS)}$, where BT (total number of bit growth) = $\log_2(nx)$ and BS (number of scales by the functions) = ceil[$\log_4(nx)-1$]. All shifts are rounded to reduce truncation noise power by 3dB.

The function takes the twiddle factors and input data, and calculates the FFT producing the frequency domain data in the y[ ] array. As the FFT allows every input point to affect every output point, which causes cache thrashing in a cache based system. This is mitigated by allowing the main FFT of size N to be divided into several steps, allowing as much data reuse as possible. For example, see the following function:

```
DSP_fft16x16r(1024,&x[0],    &w[0],    y,brev,4,    0,1024);
```

is equivalent to:

```
DSP_fft16x16r(1024,&x[2*0],  &w[0]    ,y,brev,256,  0,1024);
DSP_fft16x16r(256, &x[2*0],  &w[2*768],y,brev,4,    0,1024);
DSP_fft16x16r(256, &x[2*256],&w[2*768],y,brev,4,  256,1024);
DSP_fft16x16r(256, &x[2*512],&w[2*768],y,brev,4,  512,1024);
DSP_fft16x16r(256, &x[2*768],&w[2*768],y,brev,4,  768,1024);
```

Notice how the first FFT function is called on the entire 1K data set. It covers the first pass of the FFT until the butterfly size is 256.

The following 4 FFTs do 256-point FFTs 25% of the size. These continue down to the end when the butterfly is of size 4. They use an index to the main twiddle factor array of 0.75*2*N. This is because the twiddle factor array is composed of successively decimated versions of the main array.

N not equal to a power of 4 can be used; i.e. 512. In this case, the following would be needed to decompose the FFT:

```
DSP_fft16x16r(512, &x[0],    &w[0],    y,brev,2,    0,512);
```

is equivalent to:

```
DSP_fft16x16r(512, &x[0],    &w[0],    y,brev,128,  0,512);
DSP_fft16x16r(128, &x[2*0],  &w[2*384],y,brev,2,    0,512);
DSP_fft16x16r(128, &x[2*128],&w[2*384],y,brev,2,  128,512);
DSP_fft16x16r(128, &x[2*256],&w[2*384],y,brev,2,  256,512);
DSP_fft16x16r(128, &x[2*384],&w[2*384],y,brev,2,  384,512);
```

The twiddle factor array is composed of $\log_4(N)$ sets of twiddle factors, (3/4)*N, (3/16)*N, (3/64)*N, etc. The index into this array for each stage of the FFT is calculated by summing these indices up appropriately. For multiple FFTs, they can share the same table by calling the small FFTs from further down in the twiddle factor array, in the same way as the decomposition works for more data reuse.

Thus, the above decomposition can be summarized for a general N, radix “rad” as follows.

```
DSP_fft16x16r(N,  &x[0],      &w[0],      brev,y,N/4,0,    N)
DSP_fft16x16r(N/4,&x[0],      &w[2*3*N/4],brev,y,rad,0,    N)
DSP_fft16x16r(N/4,&x[2*N/4],  &w[2*3*N/4],brev,y,rad,N/4,  N)
DSP_fft16x16r(N/4,&x[2*N/2],  &w[2*3*N/4],brev,y,rad,N/2,  N)
DSP_fft16x16r(N/4,&x[2*3*N/4],&w[2*3*N/4],brev,y,rad,3*N/4,N)
```

As discussed previously, N can be either a power of 4 or 2. If N is a power of 4, then rad = 4, and if N is a power of 2 and not a power of 4, then rad = 2. “rad” controls how many stages of decomposition are performed. It also determines whether a radix4 or DSP_radix2 decomposition should be performed at the last stage. Hence, when “rad” is set to “N/4”, the first stage of the transform alone is performed and the code exits. To complete the FFT, four other calls are required to perform N/4 size FFTs. In fact, the ordering of these 4 FFTs amongst themselves does not matter and, thus, from a cache perspective, it helps to go through the remaining 4 FFTs in exactly the opposite order to the first. This is illustrated as follows:

```
DSP_fft16x16r(N,  &x[0],      &w[0],      brev,y,N/4,0,     N)
DSP_fft16x16r(N/4,&x[2*3*N/4],&w[2*3*N/4],brev,y,rad,3*N/4, N)
DSP_fft16x16r(N/4,&x[2*N/2],  &w[2*3*N/4],brev,y,rad,N/2,   N)
DSP_fft16x16r(N/4,&x[2*N/4],  &w[2*3*N/4],brev,y,rad,N/4,   N)
DSP_fft16x16r(N/4,&x[0],      &w[2*3*N/4],brev,y,rad,0,     N)
```

In addition, this function can be used to minimize call overhead by completing the FFT with one function call invocation as shown below:

```
DSP_fft16x16r(N, &x[0], &w[0], y, brev, rad, 0, N)
```

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void fft16x16r
(
    int             n,
    short           *ptr_x,
    short           *ptr_w,
    unsigned char   *brev,
    short           *y,
    int             radix,
    int             offset,
    int             nmax
)
```

```
{
    int   i, l0, l1, l2, h2, predj;
    int   l1p1,l2p1,h2p1, tw_offset, stride, fft_jmp;
    short xt0, yt0, xt1, yt1, xt2, yt2;
    short si1,si2,si3,co1,co2,co3;
    short xh0,xh1,xh20,xh21,xl0,xl1,xl20,xl21;
    short x_0, x_1, x_l1, x_l1p1, x_h2 , x_h2p1, x_l2, x_l2p1;
    short *x,*w;
    short *ptr_x0, *ptr_x2, *y0;
    unsigned int j, k, j0, j1, k0, k1;
    short x0, x1, x2, x3, x4, x5, x6, x7;
    short xh0_0, xh1_0, xh0_1, xh1_1;
    short xl0_0, xl1_0, xl0_1, xl1_1;
    short yt3, yt4, yt5, yt6, yt7;
    unsigned a, num;
    stride = n;         /* n is the number of complex samples */
    tw_offset = 0;
    while (stride > radix)
    {
        j = 0;
        fft_jmp = stride + (stride>>1);
        h2 = stride>>1;
        l1 = stride;
        l2 = stride + (stride>>1);
        x = ptr_x;
        w = ptr_w + tw_offset;
        for (i = 0; i < n>>1; i += 2)
        {
            co1 = w[j+0];
            si1 = w[j+1];
            co2 = w[j+2];
            si2 = w[j+3];
            co3 = w[j+4];
            si3 = w[j+5];
            j += 6;
            x_0    = x[0];
```

```
x_1    = x[1];
x_h2   = x[h2];
x_h2p1 = x[h2+1];
x_l1   = x[l1];
x_l1p1 = x[l1+1];
x_l2   = x[l2];
x_l2p1 = x[l2+1];

xh0  = x_0    + x_l1;
xh1  = x_1    + x_l1p1;
xl0  = x_0    - x_l1;
xl1  = x_1    - x_l1p1;
xh20 = x_h2   + x_l2;
xh21 = x_h2p1 + x_l2p1;
xl20 = x_h2   - x_l2;
xl21 = x_h2p1 - x_l2p1;
ptr_x0 = x;
ptr_x0[0] = ((short)(xh0 + xh20))>>1;
ptr_x0[1] = ((short)(xh1 + xh21))>>1;
ptr_x2 = ptr_x0;
x += 2;
predj = (j - fft_jmp);
if (!predj) x += fft_jmp;
if (!predj) j = 0;
xt0  = xh0 - xh20;
yt0  = xh1 - xh21;
xt1  = xl0 + xl21;
yt2  = xl1 + xl20;
xt2  = xl0 - xl21;
yt1  = xl1 - xl20;
l1p1 = l1+1;
h2p1 = h2+1;
l2p1 = l2+1;
ptr_x2[l1  ] = (xt1 * co1 + yt1 * si1 + 0x00008000) >> 16;
ptr_x2[l1p1] = (yt1 * co1 - xt1 * si1 + 0x00008000) >> 16;
ptr_x2[h2  ] = (xt0 * co2 + yt0 * si2 + 0x00008000) >> 16;
```

```
            ptr_x2[h2p1] = (yt0 * co2 - xt0 * si2 + 0x00008000) >> 16;
            ptr_x2[l2  ] = (xt2 * co3 + yt2 * si3 + 0x00008000) >> 16;
            ptr_x2[l2p1] = (yt2 * co3 - xt2 * si3 + 0x00008000) >> 16;
        }
        tw_offset += fft_jmp;
        stride = stride>>2;
    } /* end while */
    j = offset>>2;
    ptr_x0 = ptr_x;
    y0 = y;
    /* determine _norm(nmax) - 17 */
    l0 = 31;
    if (((nmax>>31)&1)==1)
        num = ~nmax;
    else
        num = nmax;
    if (!num)
        l0 = 32;
    else
    {
        a=num&0xFFFF0000; if (a) { l0-=16; num=a; }
        a=num&0xFF00FF00; if (a) { l0-= 8; num=a; }
        a=num&0xF0F0F0F0; if (a) { l0-= 4; num=a; }
        a=num&0xCCCCCCCC; if (a) { l0-= 2; num=a; }
        a=num&0xAAAAAAAA; if (a) { l0-= 1; }
    }
    l0 -= 1;
    l0 -= 17;
    if(radix == 2 || radix  == 4)
        for (i = 0; i < n; i += 4)
        {
                /* reversal computation */
                j0 = (j      ) & 0x3F;
                j1 = (j >> 6) & 0x3F;
                k0 = brev[j0];
                k1 = brev[j1];
```

```
k = (k0 << 6) |  k1;
if (l0 < 0)
  k = k << -l0;
else
  k = k >> l0;
j++;        /* multiple of 4 index */
x0   = ptr_x0[0];  x1 = ptr_x0[1];
x2   = ptr_x0[2];  x3 = ptr_x0[3];
x4   = ptr_x0[4];  x5 = ptr_x0[5];
x6   = ptr_x0[6];  x7 = ptr_x0[7];
ptr_x0 += 8;

xh0_0  = x0 + x4;
xh1_0  = x1 + x5;
xh0_1  = x2 + x6;
xh1_1  = x3 + x7;
if (radix == 2)
{
  xh0_0 = x0;
  xh1_0 = x1;
  xh0_1 = x2;
  xh1_1 = x3;
}

yt0  = xh0_0 + xh0_1;
yt1  = xh1_0 + xh1_1;
yt4  = xh0_0 - xh0_1;
yt5  = xh1_0 - xh1_1;
xl0_0  = x0 - x4;
xl1_0  = x1 - x5;
xl0_1  = x2 - x6;
xl1_1  = x3 - x7;
if (radix == 2)
{
  xl0_0 = x4;
  xl1_0 = x5;
```

```
                  xl1_1 = x6;
                  xl0_1 = x7;
                }
                yt2  = xl0_0 + xl1_1;
                yt3  = xl1_0 − xl0_1;
                yt6  = xl0_0 − xl1_1;
                yt7  = xl1_0 + xl0_1;
                if (radix == 2)
                {
                  yt7  = xl1_0 − xl0_1;
                  yt3  = xl1_0 + xl0_1;
                }
                y0[k] = yt0; y0[k+1] = yt1;
                k += n>>1;
                y0[k] = yt2; y0[k+1] = yt3;
                k += n>>1;
                y0[k] = yt4; y0[k+1] = yt5;
                k += n>>1;
                y0[k] = yt6; y0[k+1] = yt7;
        }
}
```

Special Requirements

- In-place computation is *not* allowed.
- nx must be a power of 2 or 4.
- Complex input data x[ ], twiddle factors w[ ], and output array y[ ] must be double-word aligned.
- Real values are stored in even word, imaginary in odd.
- All data are in short precision or Q.15 format. Allowed input dynamic range is $16 - (\log_2(nx) - \text{ceil}[\log_4(nx) - 1])$.
- Output results are returned in normal order.
- The FFT coefficients (twiddle factors) are generated using the program tw_fft16x16 provided in the directory ‘support\fft’. The scale factor must be 32767.5. The input data must be scaled by $2^{(\log 2(nx) - \text{ceil}[\log 4(nx) - 1])}$ to completely prevent overflow.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interruptible.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.
- A special sequence of coefficients used as generated above produces the FFT. This collapses the inner 2 loops in the traditional Burrus and Parks implementation.
- The revised FFT uses a redundant sequence of twiddle factors to allow a linear access through the data. This linear access enables data and instruction level parallelism.
- The butterfly is bit reversed; i.e. the inner 2 points of the butterfly are crossed over. This makes the data come out in bit reversed rather than in radix 4 digit reversed order. This simplifies the last pass of the loop. The BITR instruction does the bit reversal out of place.

Benchmarks

Cycles	$\text{ceil}[\log_4(nx) - 1] * (8 * nx/8 + 24) + 5.25 * nx/4 + 31$
Codesize	640 bytes

DSP_fft16x32 *Complex Forward Mixed Radix 16 x 32-bit FFT With Rounding*

Function void DSP_fft16x32(const short * restrict w, int nx, int * restrict x, int * restrict y)

Arguments

w[2*nx]	Pointer to complex Q.15 FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and 16 ≤ nx ≤ 32768.
x[2*nx]	Pointer to complex 32-bit data input.
y[2*nx]	Pointer to complex 32-bit data output.

Description This routine computes an extended precision complex forward mixed radix FFT with rounding and digit reversal. Input data x[ ] and output data y[ ] are 32-bit, coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache. The C code to generate the twiddle factors is the same as the one used for the DSP_fft16x16r routine.

Algorithm The C equivalent of the assembly code without restrictions is similar to the one shown for the DSP_fft16x16t routine. For further details, see the source code of the C version of this function, which is provided with this library. Note that the assembly code is hand optimized and restrictions may apply.

Special Requirements

❑ In-place computation is *not* allowed.

❑ The size of the FFT, nx, must be a power of 4 or 2 and greater than or equal to 16 and less than 32768.

❑ The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.

❑ The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices.

❑ The FFT coefficients (twiddle factors) are generated using the program tw_fft16x32 provided in the directory ‘support\fft’. The scale factor must be 32767.5. No scaling is done with the function; thus, the input data must be scaled by $2^{\log2(nx)}$ to completely prevent overflow.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interruptible.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.
- See the fft16x16t implementation notes, as similar ideas are used.

Benchmarks

Cycles	$(10.25 * nx/8 + 10) * \text{ceil}[\log_4(nx) - 1] + 6 * nx/4 + 81$
Codesize	1056 bytes

DSP_fft32x32 — *Complex Forward Mixed Radix 32 x 32-bit FFT With Rounding*

Function void DSP_fft32x32(const int * restrict w, int nx, int * restrict x, int * restrict y)

Arguments

w[2*nx]	Pointer to complex 32-bit FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and 16 ≤ nx ≤ 32768.
x[2*nx]	Pointer to complex 32-bit data input.
y[2*nx]	Pointer to complex 32-bit data output.

Description This routine computes an extended precision complex forward mixed radix FFT with rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 32-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache. The C code to generate the twiddle factors is similar to the one used for the DSP_fft16x16r routine, except that the factors are maintained at 32-bit precision.

Algorithm The C equivalent of the assembly code without restrictions is similar to the one shown for the DSP_fft16x16t routine. For further details, see the source code of the C version of this function, which is provided with this library. Note that the assembly code is hand optimized and restrictions may apply.

Special Requirements

- In-place computation is *not* allowed.
- The size of the FFT, nx, must be a power of 4 or 2 and greater than or equal to 16 and less than 32768.
- The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices.
- The FFT coefficients (twiddle factors) are generated using the program tw_fft32x32 provided in the directory ‘support\fft’. The scale factor must be 2147483647.5. No scaling is done with the function; thus, the input data must be scaled by $2^{\log2(nx)}$ to completely prevent overflow.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interruptible.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.
- See the fft16x16t implementation notes, as similar ideas are used.

Benchmarks

Cycles	$(12 * nx/8 + 12) * \text{ceil}[\log_4(nx) - 1] + 6 * nx/4 + 79$
Codesize	1056 bytes

DSP_fft32x32s *Complex Forward Mixed Radix 32 x 32-bit FFT With Scaling*

Function void DSP_fft32x32s(const int * restrict w, int nx, int * restrict x, int * restrict y)

Arguments

w[2*nx]	Pointer to complex 32-bit FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and 16 ≤ nx ≤ 32768.
x[2*nx]	Pointer to complex 32-bit data input.
y[2*nx]	Pointer to complex 32-bit data output.

Description This routine computes an extended precision complex forward mixed radix FFT with scaling, rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 32-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache. The C code to generate the twiddle factors is the same one used for the DSP_fft32x32 routine.

Scaling by 2 (i.e., >>1) takes place at each radix-4 stage except for the last one. A radix-4 stage can add a maximum of 2 bits, which would require scaling by 4 to completely prevent overflow. Thus, the input data must be scaled by $2^{\log2(nx)-\text{ceil}[\log4(nx)-1])}$.

Algorithm The C equivalent of the assembly code without restrictions is similar to the one shown for the fft16x16t routine. For further details, see the source code of the C version of this function, which is provided with this library. Note that the assembly code is hand optimized and restrictions may apply.

Special Requirements

- In-place computation is *not* allowed.
- The size of the FFT, nx, must be a power of 4 or 2 and greater than or equal to 16 and less than 32768.
- The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices.

- The FFT coefficients (twiddle factors) are generated using the program tw_fft32x32 provided in the directory ‘support\fft’. The scale factor must be 1073741823.5. The input data must be scaled by $2^{(\log2(nx) - \text{ceil}[\ \log4(nx)-1\ ])}$ to completely prevent overflow.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interruptible.
- Scaling is performed at each stage by shifting the results right by 1, preventing overflow.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.
- See the fft16x16t implementation notes, as similar ideas are used.

Benchmarks

Cycles	$(13 * nx/8 + 36) * \text{ceil}[\log_4(nx) - 1] + 6 * nx/4 + 36$
Codesize	928 bytes

DSP_ifft16x16 — *Complex Inverse Mixed Radix 16 x 16-bit FFT With Rounding*

Function

void DSP_ifft16x16(const short * restrict w, int nx, short * restrict x, short * restrict y)

Arguments

w[2*nx]	Pointer to complex Q.15 FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and 16 ≤ nx ≤ 32768.
x[2*nx]	Pointer to complex 16-bit data input.
y[2*nx]	Pointer to complex 16-bit data output.

Description

This routine computes a complex inverse mixed radix IFFT with rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of IFFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache.

The fft16x16 can be used to perform IFFT, by first conjugating the input, performing the FFT, and conjugating again. This allows fft16x16 to perform the IFFT as well. However, if the double conjugation needs to be avoided, then this routine uses the same twiddle factors as the FFT and performs an IFFT. The change in the sign of the twiddle factors is adjusted for in the routine. Hence, this routine uses the same twiddle factors as the fft16x16 routine.

Algorithm

The C equivalent of the assembly code without restrictions is similar to the one of the fft16x16 routine. For further details, see the source code of the C version of this function which is provided with this library.

Special Requirements

- In-place computation is *not* allowed.
- The size of the FFT, nx, must be a power of 4 or 2 and greater than or equal to 16 and less than 32768.
- The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices.
- Scaling by two is performed after each radix-4 stage except the last one.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interruptible.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.
- See the fft16x16 implementation notes, as similar ideas are used.

Benchmarks

Cycles	(6 * nx/8 + 19) * ceil[$\log_4$(nx) − 1] + 8 * nx/8 + 30
Codesize	864 bytes

DSP_ifft16x16_imre *Complex Inverse Mixed Radix 16 x 16-bit FFT With Im/Re Order*

Function void DSP_ifft16x16_imre(const short * restrict w, int nx, short * restrict x, short * restrict y)

Arguments

w[2*nx]	Pointer to complex Q.15 FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and 16 ≤ nx ≤ 32768.
x[2*nx]	Pointer to complex data input.
y[2*nx]	Pointer to complex data output.

Description This routine computes a complex inverse mixed radix IFFT with rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved imaginary and real parts. The code uses a special ordering of IFFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache.

The fft16x16_imre can be used to perform IFFT, by first conjugating the input, performing the FFT, and conjugating again. This allows fft16x16_imre to perform the IFFT as well. However, if the double conjugation needs to be avoided, then this routine uses the same twiddle factors as the FFT and performs an IFFT. The change in the sign of the twiddle factors is adjusted for in the routine. Hence, this routine uses the same twiddle factors as the fft16x16_imre routine.

Algorithm The C equivalent of the assembly code without restrictions is similar to the one of the ifft16x16 routine. For further details, see the source code of the C version of this function which is provided with this library.

Special Requirements

❑ In-place computation is *not* allowed.

❑ The size of the FFT, nx, must be a power of 4 or 2 and greater than or equal to 16 and less than 32768.

❑ The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.

❑ The input and output data are complex, with the **imaginary/real** components stored in adjacent locations in the array. The imaginary components are stored at even array indices, and the real components are stored at odd array indices.

❑ Scaling by two is performed after each radix-4 stage except the last one.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interruptible.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.
- See the fft16x16 implementation notes, as similar ideas are used.

Benchmarks

Cycles	$(6 * nx/8 + 19) * \mathrm{ceil}[\log_4(nx) - 1] + 8 * nx/8 + 30$
Codesize	864 bytes

DSP_ifft16x32 *Complex Inverse Mixed Radix 16 x 32-bit FFT With Rounding*

Function void DSP_ifft16x32(const short * restrict w, int nx, int * restrict x, int * restrict y)

Arguments

w[2*nx]	Pointer to complex Q.15 FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and 16 ≤ nx ≤ 32768.
x[2*nx]	Pointer to complex 32-bit data input.
y[2*nx]	Pointer to complex 32-bit data output.

Description This routine computes an extended precision complex inverse mixed radix FFT with rounding and digit reversal. Input data x[ ] and output data y[ ] are 32-bit, coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache.

fft16x32 can be reused to perform IFFT, by first conjugating the input, performing the FFT, and conjugating again. This allows fft16x32 to perform the IFFT as well. However, if the double conjugation needs to be avoided, then this routine uses the same twiddle factors as the FFT and performs an IFFT. The change in the sign of the twiddle factors is adjusted for in the routine. Hence, this routine uses the same twiddle factors as the fft16x32 routine.

Algorithm The C equivalent of the assembly code without restrictions is similar to the one shown for the fft16x16t routine. For further details, see the source code of the C version of this function which is provided with this library. Note that the assembly code is hand optimized and restrictions may apply.

Special Requirements

- In-place computation is *not* allowed.
- The size of the FFT, nx, must be a power of 4 or 2 and greater than or equal to 16 and less than 32768.
- The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices.

- The FFT coefficients (twiddle factors) are generated using the program tw_fft16x32 provided in the directory 'support\fft'. The scale factor must be 32767.5. No scaling is done with the function; thus the input data must be scaled by $2^{\log2(nx)}$ to completely prevent overflow.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interruptible.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.
- See the fft16x16t implementation notes, as similar ideas are used.

Benchmarks

Cycles	$(12.5 * nx/8 + 30) * \text{ceil}[\log_4(nx) - 1] + 6 * nx/4 + 32$
Codesize	864 bytes

DSP_ifft32x32 *Complex Inverse Mixed Radix 32 x 32-bit FFT With Rounding*

Function void DSP_ifft32x32(const int * restrict w, int nx, int * restrict x, int * restrict y)

Arguments

w[2*nx]	Pointer to complex 32-bit FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4, and $16 \le nx \le 32768$.
x[2*nx]	Pointer to complex 32-bit data input.
y[2*nx]	Pointer to complex 32-bit data output.

Description This routine computes an extended precision complex inverse mixed radix FFT with rounding and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 32-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache.

fft32x32 can be reused to perform IFFT, by first conjugating the input, performing the FFT, and conjugating again. This allows fft32x32 to perform the IFFT as well. However, if the double conjugation needs to be avoided, then this routine uses the same twiddle factors as the FFT and performs an IFFT. The change in the sign of the twiddle factors is adjusted for in the routine. Hence, this routine uses the same twiddle factors as the fft32x32 routine.

Algorithm The C equivalent of the assembly code without restrictions is similar to the one shown for the fft16x16t routine. For further details, see the source code of the C version of this function which is provided with this library. Note that the assembly code is hand optimized and restrictions may apply.

Special Requirements

- In-place computation is *not* allowed.
- The size of the IFFT, nx, must be a power of 4 or 2 and greater than or equal to 16 and less than 32768.
- The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices.

- The FFT coefficients (twiddle factors) are generated using the program tw_fft32x32 provided in the directory 'support\fft'. The scale factor must be 2147483647.5. No scaling is done with the function; thus the input data must be scaled by $2^{\log2(nx)}$ to completely prevent overflow.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interruptible.
- The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform.
- See the fft16x16t implementation notes, as similar ideas are used.

Benchmarks

Cycles	$(13*nx/8 + 28) * \text{ceil}(\log_4(nx) - 1) + 6 * nx/4 + 39$
Codesize	960 bytes

4.4 Filtering and Convolution

DSP_fir_cplx *Complex FIR Filter*

Function void DSP_fir_cplx (const short * restrict x, const short * restrict h, short * restrict r, int nh, int nr)

Arguments

x[2*(nr+nh−1)]	Complex input data. x must point to x[2*(nh−1)].
h[2*nh]	Complex coefficients (in normal order).
r[2*nr]	Complex output data.
nh	Number of complex coefficients. Must be a multiple of 2.
nr	Number of complex output samples. Must be a multiple of 4.

Description This function implements the FIR filter for complex input data. The filter has nr output samples and nh coefficients. Each array consists of an even and odd term with even terms representing the real part and the odd terms the imaginary part of the element. The pointer to input array x must point to the (nh)th complex sample; i.e., element 2*(nh−1), upon entry to the function. The coefficients are expected in normal order.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_fir_cplx(short *x, short *h, short *r,short nh, short
nr)
{
    short i,j;
    int imag, real;
    for (i = 0; i < 2*nr; i += 2){
        imag = 0;
        real = 0;
        for (j = 0; j < 2*nh; j += 2){
            real += h[j] * x[i−j] − h[j+1] * x[i+1−j];
            imag += h[j] * x[i+1−j] + h[j+1] * x[i−j];
        }
        r[i] = (real >> 15);
        r[i+1] = (imag >> 15);
    }
}
```

Special Requirements

- The number of coefficients nh must be a multiple of 2.
- The number of output samples nr must be a multiple of 4.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- The outer loop is unrolled 4 times while the inner loop is not unrolled.
- Both inner and outer loops are collapsed in one loop.
- ADDAH and SUBAH are used along with PACKH2 to perform accumulation, shift, and data packing.
- Collapsed one stage of epilog and prolog each.

Benchmarks

Cycles	nr * nh/2 + 7
Codesize	448 bytes

DSP_fir_cplx_hM4X4 *Complex FIR Filter*

Function void DSP_fir_cplx _hM4X4(const short * restrict x, const short * restrict h, short * restrict r, int nh, int nr)

Arguments

x[2*(nr+nh−1)]	Complex input data. x must point to x[2*(nh−1)].
h[2*nh]	Complex coefficients (in normal order).
r[2*nr]	Complex output data.
nh	Number of complex coefficients. Must be a multiple of 4.
nr	Number of complex output samples. Must be a multiple of 4.

Description This function implements the FIR filter for complex input data. The filter has nr output samples and nh coefficients. Each array consists of an even and odd term with even terms representing the real part and the odd terms the imaginary part of the element. The pointer to input array x must point to the (nh)th complex sample; i.e., element 2*(nh−1), upon entry to the function. The coefficients are expected in normal order.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_fir_cplx(short *x, short *h, short *r,short nh, short
nr)
{
    short i,j;
    int imag, real;
    for (i = 0; i < 2*nr; i += 2){
        imag = 0;
        real = 0;
        for (j = 0; j < 2*nh; j += 2){
            real += h[j] * x[i−j] − h[j+1] * x[i+1−j];
            imag += h[j] * x[i+1−j] + h[j+1] * x[i−j];
        }
        r[i] = (real >> 15);
        r[i+1] = (imag >> 15);
    }
}
```

Special Requirements

- The number of coefficients nh must be larger or equal to 4 and a multiple of 4.
- The number of output samples nr must be a multiple of 4.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is fully interruptible.
- The outer loop is unrolled 4 times while the inner loop is not unrolled.
- Both inner and outer loops are collapsed in one loop.
- ADDAH and SUBAH are used along with PACKH2 to perform accumulation, shift and data packing.
- Collapsed one stage of epilog and prolog each.

Benchmarks

Cycles	nr * nh*9/16 + 40
Codesize	384 bytes

DSP_fir_gen *FIR Filter*

Function void DSP_fir_gen (const short * restrict x, const short * restrict h, short * restrict r, int nh, int nr)

Arguments

x[nr+nh−1]	Pointer to input array of size nr + nh − 1.
h[nh]	Pointer to coefficient array of size nh (coefficients must be in reverse order).
r[nr]	Pointer to output array of size nr. Must be word aligned.
nh	Number of coefficients. Must be ≥5.
nr	Number of samples to calculate. Must be a multiple of 4.

Description Computes a real FIR filter (direct-form) using coefficients stored in vector h[ ]. The real data input is stored in vector x[ ]. The filter output result is stored in vector r[ ]. It operates on 16-bit data with a 32-bit accumulate. The filter calculates nr output samples using nh coefficients.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_fir_gen(short *x, short *h, short *r, int nh, int nr)
{
    int i, j, sum;

    for (j = 0; j < nr; j++) {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
    }
}
```

Special Requirements

- The number of coefficients, nh, must be greater than or equal to 5. Coefficients must be in reverse order.
- The number of outputs computed, nr, must be a multiple of 4 and greater than or equal to 4.
- Array r[ ] must be word aligned.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- Load double-word instruction is used to simultaneously load four values in a single clock cycle.
- The inner loop is unrolled four times and will always compute a multiple of 4 of nh and nr. If nh is not a multiple of 4, the code will fill in zeros to make nh a multiple of 4.
- This code yields best performance when the ratio of outer loop to inner loop is less than or equal to 4.

Benchmarks

Cycles: Not available
Codesize: Not available

DSP_fir_gen_hM17_rA8X8 *FIR Filter*

Function void DSP_fir_gen_hM17_rA8X8 (const short * restrict x, const short * restrict h, short * restrict r, int nh, int nr)

Arguments

x[nr+nh−1]	Pointer to input array of size nr + nh − 1.
h[nh]	Pointer to coefficient array of size nh (coefficients must be in reverse order).
r[nr]	Pointer to output array of size nr. Must be double word aligned.
nh	Number of coefficients. Must be ≥17.
nr	Number of samples to calculate. Must be a multiple of 8.

Description Computes a real FIR filter (direct-form) using coefficients stored in vector h[ ]. The real data input is stored in vector x[ ]. The filter output result is stored in vector r[ ]. It operates on 16-bit data with a 32-bit accumulate. The filter calculates nr output samples using nh coefficients.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_fir_gen(short *x, short *h, short *r, int nh, int nr)
{
    int i, j, sum;

    for (j = 0; j < nr; j++) {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
    }
}
```

Special Requirements

- The number of coefficients, nh, must be greater than or equal to 17. Coefficients must be in reverse order.
- The number of outputs computed, nr, must be a multiple of 8 and greater than or equal to 8.
- Array r[ ] must be word aligned.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is fully interruptible.
- Load double-word instruction is used to simultaneously load four values in a single clock cycle.
- The inner loop is unrolled four times and will always compute a multiple of 4 of nh and nr. If nh is not a multiple of 4, the code will fill in zeros to make nh a multiple of 4.
- This code yields best performance when the ratio of outer loop to inner loop is less than or equal to 4.

Benchmarks

Cycles: 3*ceil(nh/4)*nr/4+39
Codesize: 416 bytes

DSP_fir_r4 *FIR Filter (when the number of coefficients is a multiple of 4)*

Function void DSP_fir_r4 (const short * restrict x, const short * restrict h, short * restrict r, int nh, int nr)

Arguments

x[nr+nh−1]	Pointer to input array of size nr + nh – 1.
h[nh]	Pointer to coefficient array of size nh (coefficients must be in reverse order).
r[nr]	Pointer to output array of size nr.
nh	Number of coefficients. Must be multiple of 4 and ≥8.
nr	Number of samples to calculate. Must be multiple of 4.

Description Computes a real FIR filter (direct-form) using coefficients stored in vector h[ ]. The real data input is stored in vector x[ ]. The filter output result is stored in vector r[ ]. This FIR operates on 16-bit data with a 32-bit accumulate. The filter calculates nr output samples using nh coefficients.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_fir_r4(short *x, short *h, short *r, int nh, int nr)
{
    int i, j, sum;

    for (j = 0; j < nr; j++) {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
    }
}
```

Special Requirements

- The number of coefficients, nh, must be a multiple of 4 and greater than or equal to 8. Coefficients must be in reverse order.
- The number of outputs computed, nr, must be a multiple of 4 and greater than or equal to 4.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- The load double-word instruction is used to simultaneously load four values in a single clock cycle.
- The inner loop is unrolled four times and will always compute a multiple of 4 output samples.

Benchmarks

Cycles	(8 + nh) * nr/4 + 9
Codesize	308 bytes

DSP_fir_r8 *FIR Filter (when the number of coefficients is a multiple of 8)*

Function void DSP_fir_r8_hM16_rM8A8X8 (short *x, short *h, short *r, int nh, int nr)

Arguments

x[nr+nh−1]	Pointer to input array of size nr + nh – 1.
h[nh]	Pointer to coefficient array of size nh (coefficients must be in reverse order).
r[nr]	Pointer to output array of size nr. Must be word aligned.
nh	Number of coefficients. Must be multiple of 8, ≥ 8.
nr	Number of samples to calculate. Must be multiple of 4.

Description Computes a real FIR filter (direct-form) using coefficients stored in vector h[ ]. The real data input is stored in vector x[ ]. The filter output result is stored in vector r[ ]. This FIR operates on 16-bit data with a 32-bit accumulate. The filter calculates nr output samples using nh coefficients.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_fir_r8 (short *x, short *h, short *r, int nh, int nr)
{
    int i, j, sum;

    for (j = 0; j < nr; j++) {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
        }
}
```

Special Requirements

- ❑ The number of coefficients, nh, must be a multiple of 8 and greater than or equal to 8. Coefficients must be in reverse order.
- ❑ The number of outputs computed, nr, must be a multiple of 4 and greater than or equal to 4.
- ❑ Array r[ ] must be word aligned.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interruptible.
- The load double-word instruction is used to simultaneously load four values in a single clock cycle.
- The inner loop is unrolled 4 times and will always compute a multiple of 4 output samples.
- The outer loop is conditionally executed in parallel with the inner loop. This allows for a zero overhead outer loop.

Benchmarks

Cycles	nh*nr/4 + 17
Codesize	336 bytes

DSP_fir_r8_hM16_rM8A8X8 *FIR Filter (the number of coefficients is a multiple of 8)*

Function void DSP_fir_r8_hM16_rM8A8X8 (short *x, short *h, short *r, int nh, int nr)

Arguments

x[nr+nh−1]	Pointer to input array of size nr + nh – 1.
h[nh]	Pointer to coefficient array of size nh (coefficients must be in reverse order).
r[nr]	Pointer to output array of size nr. Must be double word aligned.
nh	Number of coefficients. Must be multiple of 8, ≥ 16.
nr	Number of samples to calculate. Must be multiple of 8, ≥.8.

Description Computes a real FIR filter (direct-form) using coefficients stored in vector h[ ]. The real data input is stored in vector x[ ]. The filter output result is stored in vector r[ ]. This FIR operates on 16-bit data with a 32-bit accumulate. The filter calculates nr output samples using nh coefficients.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_fir_r8 (short *x, short *h, short *r, int nh, int nr)
{
    int i, j, sum;

    for (j = 0; j < nr; j++) {
        sum = 0;
        for (i = 0; i < nh; i++)
            sum += x[i + j] * h[i];
        r[j] = sum >> 15;
        }
}
```

Special Requirements

- The number of coefficients, nh, must be a multiple of 8 and greater than or equal to 16. Coefficients must be in reverse order.
- The number of outputs computed, nr, must be a multiple of 8 and greater than or equal to 8.
- Array r[ ] must be double word aligned.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interruptible.
- The load double-word instruction is used to simultaneously load four values in a single clock cycle.
- The inner loop is unrolled 4 times and will always compute a multiple of 4 output samples.
- The outer loop is conditionally executed in parallel with the inner loop. This allows for a zero overhead outer loop.

Benchmarks	Cycles	When nh>32, nh*nr/8+22
		Otherwise, 32*nr/8+22
	Codesize	640 bytes

DSP_fir_sym *Symmetric FIR Filter*

Function void DSP_fir_sym (const short * restrict x, const short * restrict h, short * restrict r, int nh, int nr, int s)

Arguments

x[nr+2*nh]	Pointer to input array of size nr + 2*nh. Must be double-word aligned.
h[nh+1]	Pointer to coefficient array of size nh + 1. Coefficients are in normal order and only half (nh+1 out of 2*nh+1) are required. Must be double-word aligned.
r[nr]	Pointer to output array of size nr. Must be word aligned.
nh	Number of coefficients. Must be multiple of 8. The number of original symmetric coefficients is 2*nh+1.
nr	Number of samples to calculate. Must be multiple of 4.
s	Number of insignificant digits to truncate; e.g., 15 for Q.15 input data and coefficients.

Description This function applies a symmetric filter to the input samples. The filter tap array h[] provides ‘nh+1’ total filter taps. The filter tap at h[nh] forms the center point of the filter. The taps at h[nh − 1] through h[0] form a symmetric filter about this central tap. The effective filter length is thus 2*nh+1 taps.

The filter is performed on 16-bit data with 16-bit coefficients, accumulating intermediate results to 40-bit precision. The accumulator is rounded and truncated according to the value provided in ‘s’. This allows a variety of Q-points to be used.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_fir_sym(short *x, short *h, short *r, int nh, int nr,
int s)
{
    int             i, j;
    long            y0;
    long            round = (long) 1 << (s - 1);
    for (j = 0; j < nr; j++) {
        y0 = round;
        for (i = 0; i < nh; i++)
```

```
            y0 += (short) (x[j + i] + x[j + 2 * nh – i]) * h[i];
        y0 += x[j + nh] * h[nh];
        r[j] = (int) (y0 >> s);
    }
}
```

Special Requirements

- ❑ nh must be a multiple of 8. The number of original symmetric coefficients is 2*nh+1. Only half (nh+1) are required.
- ❑ nr must be a multiple of 4.
- ❑ x[ ] and h[ ] must be double-word aligned.
- ❑ r[ ] must be word aligned.

Implementation Notes

- ❑ ***Bank Conflicts:*** No bank conflicts occur.
- ❑ ***Interruptibility:*** The code is interruptible.
- ❑ The load double-word instruction is used to simultaneously load four values in a single clock cycle.
- ❑ The inner loop is unrolled eight times.

Benchmarks

Cycles	(10 * nh/8 + 15) * nr/4 + 26
Codesize	664 bytes

DSP_iir *IIR With 5 Coefficients*

Function void DSP_iir (short * restrict r1, const short * restrict x, short * restrict r2, const short * restrict h2, const short * restrict h1, int nr)

Arguments

r1[nr+4]	Output array (used in actual computation. First four elements must have the previous outputs.)
x[nr+4]	Input array
r2[nr]	Output array (stored)
h2[5]	Moving-average filter coefficients
h1[5]	Auto-regressive filter coefficients. h1[0] is not used.
nr	Number of output samples. Must be ≥ 8.

Description The IIR performs an auto-regressive moving-average (ARMA) filter with 4 auto-regressive filter coefficients and 5 moving-average filter coefficients for nr output samples.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_iir(short *r1, short *x, short *r2, short *h2,
short *h1, int nr)
{
    int j,i;
    int sum;
    for (i=0; i<nr; i++){
        sum = h2[0] * x[4+i];
        for (j = 1; j <= 4; j++)
            sum += h2[j]*x[4+i–j]–h1[j]*r1[4+i–j];
        r1[4+i] = (sum >> 15);
        r2[i] = r1[4+i];
    }
}
```

Special Requirements

- nr is greater than or equal to 8.
- Input data array x[ ] contains nr + 4 input samples to produce nr output samples.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- Output array r1[ ] contains nr + 4 locations, r2[ ] contains nr locations for storing nr output samples. The output samples are stored with an offset of 4 into the r1[ ] array.
- The inner loop that iterated through the filter coefficients is completely unrolled.

Benchmarks

Cycles	4 * nr + 21
Codesize	276 bytes

DSP_iirlat *All-Pole IIR Lattice Filter*

Function void DSP_iirlat(const short * restrict x, int nx, const short * restrict k, int nk, int * restrict b, short * restrict r)

Arguments

x[nx]	Input vector (16-bit).
nx	Length of input vector.
k[nk]	Reflection coefficients in Q.15 format.
nk	Number of reflection coefficients/lattice stages. Must be >=4. Make multiple of 2 to avoid bank conflicts.
b[nk+1]	Delay line elements from previous call. Should be initialized to all zeros prior to the first call.
r[nx]	Output vector (16-bit).

Description This routine implements a real all-pole IIR filter in lattice structure (AR lattice). The filter consists of nk lattice stages. Each stage requires one reflection coefficient k and one delay element b. The routine takes an input vector x[] and returns the filter output in r[]. Prior to the first call of the routine, the delay elements in b[] should be set to zero. The input data may have to be pre-scaled to avoid overflow or achieve better SNR. The reflections coefficients lie in the range −1.0 < k < 1.0. The order of the coefficients is such that k[nk−1] corresponds to the first lattice stage after the input and k[0] corresponds to the last stage.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
   void iirlat(short *x, int nx, short *k, int nk, int *b,
short *r)
{
    int rt;     /* output       */
    int i, j;

    for (j=0; j<nx; j++)
    {
        rt = x[j] << 15;
        for (i = nk - 1; i >= 0; i--)
        {
            rt       = rt   - (short)(b[i] >> 15) * k[i];
            b[i + 1] = b[i] + (short)(rt   >> 15) * k[i];
        }
        b[0] = rt;
        r[j] = rt >> 15;
    }
}
```

Special Requirements

- nk must be >= 4.
- No special alignment requirements
- See *Bank Conflicts* for avoiding bank conflicts

Implementation Notes

- ***Bank Conflicts:*** nk should be a multiple of 2, otherwise bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- Prolog and epilog of the inner loop are partially collapsed and overlapped to reduce outer loop overhead.

Benchmarks

Cycles	(2 * nk + 7) * nx + 9	(without bank conflicts)
Codesize	352 bytes	

4.5 Math

DSP_dotp_sqr *Vector Dot Product and Square*

Function

int DSP_dotp_sqr(int G, const short * restrict x, const short * restrict y, int * restrict r, int nx)

Arguments

G	Calculated value of G (used in the VSELP coder).
x[nx]	First vector array
y[nx]	Second vector array
r	Result of vector dot product of x and y.
nx	Number of elements. Must be multiple of 4, and ≥12.
return int	New value of G.

Description

This routine performs an nx element dot product of x[ ] and y[ ] and stores it in r. It also squares each element of y[ ] and accumulates it in G. G is passed back to the calling function in register A4. This computation of G is used in the VSELP coder.

Algorithm

This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
int DSP_dotp_sqr (int G,short *x,short *y,int *r,
int nx)
{
    short *y2;
    short *endPtr2;
    y2 = x;
    for (endPtr2 = y2 + nx; y2 < endPtr2; y2++){
        *r += *y * *y2;
        G += *y * *y;
        y++;
    }
    return(G);
}
```

Special Requirements nx must be a multiple of 4 and greater than or equal to 12.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx/2 + 21
Codesize	128

DSP_dotprod *Vector Dot Product*

Function int DSP_dotprod(const short * restrict x, const short * restrict y, int nx)

Arguments

x[nx]	First vector array. Must be double-word aligned.
y[nx]	Second vector array. Must be double word-aligned.
nx	Number of elements of vector. Must be multiple of 4.
return int	Dot product of x and y.

Description This routine takes two vectors and calculates their dot product. The inputs are 16-bit short data and the output is a 32-bit number.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
int DSP_dotprod(short x[ ],short y[ ], int nx)
{
    int sum;
    int i;
    sum = 0;
    for(i=0; i<nx; i++){
        sum += (x[i] * y[i]);
    }
    return (sum);
}
```

Special Requirements

- ❑ The input length must be a multiple of 4.
- ❑ The input data x[ ] and y[ ] are stored on double-word aligned boundaries.
- ❑ To avoid bank conflicts, the input arrays x[ ] and y[ ] must be offset by 4 half-words (8 bytes).

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur if the input arrays x[ ] and y[ ] are offset by 4 half-words (8 bytes).
- ***Interruptibility:*** The code is fully interruptible.
- The code is unrolled 4 times to enable full memory and multiplier bandwidth to be utilized.
- Interrupts are masked by branch delay slots only.
- Prolog collapsing has been performed to reduce codesize.

Benchmarks

Cycles	nx / 4 + 14
Codesize	64 bytes

DSP_maxval *Maximum Value of Vector*

Function short DSP_maxval (const short *x, int nx)

Arguments

x[nx]	Pointer to input vector of size nx.
nx	Length of input data vector. Must be multiple of 8 and ≥32.
return short	Maximum value of a vector.

Description This routine finds the element with maximum value in the input vector and returns that value.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
short DSP_maxval(short x[ ], int nx)
{
    int i, max;
    max = −32768;

    for (i = 0; i < nx; i++)
        if (x[i] > max)
            max = x[i];
    return max;
}
```

Special Requirements nx is a multiple of 8 and greater than or equal to 32.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx / 4 + 10
Codesize	116 bytes

DSP_maxidx *Index of Maximum Element of Vector*

Function int DSP_maxidx (const short *x, int nx)

Arguments

x[nx]	Pointer to input vector of size nx. Must be double-word aligned.
nx	Length of input data vector. Must be multiple of 16 and ≥ 48.
return int	Index for vector element with maximum value.

Description This routine finds the max value of a vector and returns the index of that value.

The input array is treated as 16 separate columns that are interleaved throughout the array. If values in different columns are equal to the maximum value, then the element in the leftmost column is returned. If two values within a column are equal to the maximum, then the one with the lower index is returned. Column takes precedence over index.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
int DSP_maxidx(short x[ ], int nx)
{
    int max, index, i;
    max = −32768;
    for (i = 0; i < nx; i++)
        if (x[i] > max) {
            max = x[i];
            index = i;
        }
    return index;
}
```

Special Requirements

- nx must be a multiple of 16 and greater than or equal to 48.
- The input vector x[ ] must be double-word aligned.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- The code is unrolled 16 times to enable the full bandwidth of LDDW and MAX2 instructions to be utilized. This splits the search into 16 sub-ranges. The global maximum is then found from the list of maximums of the sub-ranges. Then, using this offset from the sub-ranges, the global maximum and the index of it are found using a simple match. For common maximums in multiple ranges, the index will be different to the above C code.
- This code requires 40 bytes of stack space for a temporary buffer.

Benchmarks

Cycles	5 * nx / 16 + 42
Codesize	388 bytes

DSP_minval *Minimum Value of Vector*

Function short DSP_minval (const short *x, int nx)

Arguments

x [nx]	Pointer to input vector of size nx.
nx	Length of input data vector. Must be multiple of 4 and ≥20.
return short	Maximum value of a vector.

Description This routine finds the minimum value of a vector and returns the value.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
short DSP_minval(short x[ ], int nx)
{
   int i, min;
   min = 32767;

   for (i = 0; i < nx; i++)
      if (x[i] < min)
         min = x[i];
   return min;
}
```

Special Requirements nx is a multiple of 4 and greater than or equal to 20.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- The input data is loaded using double word wide loads, and the MIN2 instruction is used to get to the minimum.

Benchmarks

Cycles	nx / 4 +10
Codesize	116 bytes

DSP_mul32 *32-Bit Vector Multiply*

Function void DSP_mul32(const int * restrict x, const int * restrict y, int * restrict r, short nx)

Arguments

x[nx]	Pointer to input data vector 1 of size nx. Must be double-word aligned.
y[nx]	Pointer to input data vector 2 of size nx. Must be double-word aligned.
r[nx]	Pointer to output data vector of size nx. Must be double-word aligned.
nx	Number of elements in input and output vectors. Must be multiple of 8 and ≥16.

Description The function performs a Q.31 x Q.31 multiply and returns the upper 32 bits of the result. The result of the intermediate multiplies are accumulated into a 40-bit long register pair, as there could be potential overflow. The contribution of the multiplication of the two lower 16-bit halves are not considered. The output is in Q.30 format. Results are accurate to least significant bit.

Algorithm In the comments below, X and Y are the two input values. Xhigh and Xlow represent the upper and lower 16 bits of X. This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_mul32(const int *x, const int *y, int *r,
short nx)
{
    short     i;
    int       a,b,c,d,e;
    for(i=nx;i>0;i--)
    {
        a=*(x++);
        b=*(y++);
        c=_mpyluhs(a,b); /* Xlow*Yhigh */
        d=_mpyhslu(a,b); /* Xhigh*Ylow */
        e=_mpyh(a,b);  /* Xhigh*Yhigh */
        d+=c;            /* Xhigh*Ylow+Xlow*Yhigh */
        d=d>>16;    /* (Xhigh*Ylow+Xlow*Yhigh)>>16 */
        e+=d;       /* Xhigh*Yhigh + */
                    /* (Xhigh*Ylow+Xlow*Yhigh)>>16 */
        *(r++)=e;
    }
}
```

Special Requirements

- ❑ nx must be a multiple of 8 and greater than or equal to 16.
- ❑ Input and output vectors must be double-word aligned.

Implementation Notes

- ❑ ***Bank Conflicts:*** No bank conflicts occur.
- ❑ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- ❑ The MPYHI instruction is used to perform 16 x 32 multiplies to form 48-bit intermediate results.

Benchmarks

Cycles	9 * nx/8 + 18
Codesize	512 bytes

DSP_neg32 — *32-Bit Vector Negate*

Function void DSP_neg32(int *x, int *r, short nx)

Arguments

x[nx]	Pointer to input data vector 1 of size nx with 32-bit elements. Must be double-word aligned.
r[nx]	Pointer to output data vector of size nx with 32-bit elements. Must be double-word aligned.
nx	Number of elements of input and output vectors. Must be a multiple of 4 and ≥8.

Description This function negates the elements of a vector (32-bit elements). The input and output arrays must not be overlapped except for where the input and output pointers are exactly equal.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_neg32(int *x, int *r, short nx)
{
    short i;
    for(i=nx; i>0; i--)
        *(r++)=-*(x++);
}
```

Special Requirements

- nx must be a multiple of 4 and greater than or equal to 8.
- The arrays x[ ] and r[ ] must be double-word aligned.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- The loop is unrolled twice and pipelined.

Benchmarks

Cycles	nx/2 + 19
Codesize	124 bytes

DSP_recip16 *16-Bit Reciprocal*

Function void DSP_recip16 (short *x, short *rfrac, short *rexp, short nx)

Arguments

x[nx]	Pointer to Q.15 input data vector of size nx.
rfrac[nx]	Pointer to Q.15 output data vector for fractional values.
rexp[nx]	Pointer to output data vector for exponent values.
nx	Number of elements of input and output vectors.

Description This routine returns the fractional and exponential portion of the reciprocal of an array x[ ] of Q.15 numbers. The fractional portion rfrac is returned in Q.15 format. Since the reciprocal is always greater than 1, it returns an exponent such that:

```
(rfrac[i] * 2rexp[i]) = true reciprocal
```

The output is accurate up to the least significant bit of rfrac, but note that this bit could carry over and change rexp. For a reciprocal of 0, the procedure will return a fractional part of 7FFFh and an exponent of 16.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_recip16(short *x, short *rfrac, short *rexp, short
nx)
{
    int i,j,a,b;
    short neg, normal;
    for(i=nx; i>0; i--)
    {
        a=*(x++);
        if(a<0)             /* take absolute value */
        {
            a=-a;
            neg=1;
        }
        else neg=0;
        normal=_norm(a);    /* normalize number */
        a=a<<normal;
        *(rexp++)=normal-15;   /* store exponent */
        b=0x80000000;       /* dividend = 1 */
        for(j=15;j>0;j--)
            b=_subc(b,a);    /* divide */
        b=b&0x7FFF;         /* clear remainder
                            /* (clear upper half) */
        if(neg) b=-b;    /* if originally
                            /* negative, negate */
        *(rfrac++)=b;    /* store fraction */
    }
}
```

Special Requirements None

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interruptible.
- The conditional subtract instruction, SUBC, is used for division. SUBC is used once for every bit of quotient needed (15).

Benchmarks

	Cycles	8 * nx + 14
	Codesize	196 bytes

DSP_vecsumsq *Sum of Squares*

Function int DSP_vecsumsq (const short *x, int nx)

Arguments

x[nx]	Input vector
nx	Number of elements in x. Must be multiple of 4 and ≥8.
return int	Sum of the squares

Description This routine returns the sum of squares of the elements contained in the vector x[ ].

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
int DSP_vecsumsq(short x[ ], int nx)
{
    int i, sum=0;

    for(i=0; i<nx; i++)
    {
        sum += x[i]*x[i];
    }
    return(sum);
}
```

Special Requirements nx must be a multiple of 4 and greater than or equal to 32.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- The code is unrolled 4 times to enable full memory and multiplier bandwidth to be utilized.

Benchmarks

Cycles	nx/4 + 11
Codesize	188 bytes

DSP_w_vec — *Weighted Vector Sum*

Function void DSP_w_vec(const short * restrict x, const short * restrict y, short m, short * restrict r, short nr)

Arguments

x[nr]	Vector being weighted. Must be double-word aligned.
y[nr]	Summation vector. Must be double-word aligned.
m	Weighting factor
r[nr]	Output vector
nr	Dimensions of the vectors. Must be multiple of 8 and ≥8.

Description This routine is used to obtain the weighted vector sum. Both the inputs and output are 16-bit numbers.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_w_vec(short x[ ],short y[ ],short m,
short r[ ],short nr)
{
    short i;

    for (i=0; i<nr; i++) {
        r[i] = ((m * x[i]) >> 15) + y[i];
    }
}
```

Special Requirements

- nr must be a multiple of 8 and ≥ 8.
- Vectors x[ ] and y[ ] must be double-word aligned.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- Input is loaded in double-words.
- Use of packed data processing to sustain throughput.

Benchmarks

Cycles	3 * nr/8 + 18
Codesize	144 bytes

4.6 Matrix

DSP_mat_mul *Matrix Multiplication*

Function void DSP_mat_mul(const short * restrict x, int r1, int c1, const short * restrict y, int c2, short * restrict r, int qs)

Arguments

x [r1*c1]	Pointer to input matrix of size r1*c1.
r1	Number of rows in matrix x.
c1	Number of columns in matrix x. Also number of rows in y.
y [c1*c2]	Pointer to input matrix of size c1*c2.
c2	Number of columns in matrix y.
r [r1*c2]	Pointer to output matrix of size r1*c2.
qs	Final right−shift to apply to the result.

Description This function computes the expression “r = x * y” for the matrices x and y. The columnar dimension of x must match the row dimension of y. The resulting matrix has the same number of rows as x and the same number of columns as y.

The values stored in the matrices are assumed to be fixed-point or integer values. All intermediate sums are retained to 32-bit precision, and no overflow checking is performed. The results are right-shifted by a user-specified amount, and then truncated to 16 bits.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_mat_mul(short *x, int r1, int c1, short *y, int c2,
short *r, int qs)
{
    int i, j, k;
    int sum;

    /* -------------------------------------------------- */
    /*  Multiply each row in x by each column in y.  The  */
    /*  product of row m in x and column n in y is placed */
    /*  in position (m,n) in the result.                  */
    /* -------------------------------------------------- */
```

```
        for (i = 0; i < r1; i++)
            for (j = 0; j < c2; j++)
            {
                sum = 0;

                for (k = 0; k < c1; k++)
                    sum += x[k + i*c1] * y[j + k*c2];

                r[j + i*c2] = sum >> qs;
            }
}
```

Special Requirements

- The arrays x[], y[], and r[] are stored in distinct arrays. That is, in-place processing is not allowed.
- The input matrices have minimum dimensions of at least 1 row and 1 column, and maximum dimensions of 32767 rows and 32767 columns.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** This code blocks interrupts during its innermost loop. Interrupts are not blocked otherwise. As a result, interrupts can be blocked for up to 0.25*c1’ + 16 cycles at a time.
- The ‘i’ loop and ‘k’ loops are unrolled 2x. The ’j’ loop is unrolled 4x. For dimensions that are not multiples of the various loops’ unroll factors, this code calculates extra results beyond the edges of the matrix. These extra results are ultimately discarded. This allows the loops to be unrolled for efficient operation on large matrices while not losing flexibility.

Benchmarks

Cycles 0.25 * (r1’ * c2’ * c1’) + 2.25 * (r1’ * c2’) + 11, where:
r1’ = 2 * ceil(r1/2.0) (r1 rounded up to next even)
c1’ = 2 * ceil(c1/2.0) (c1 rounded up to next even)
c2’ = 4 * ceil(c2/4.0) (c2 rounded up to next mult of 4)
For r1= 1, c1= 1, c2= 1: 33 cycles
For r1= 8, c1=20, c2= 8: 475 cycles

Codesize 416 bytes

DSP_mat_trans *Matrix Transpose*

Function void DSP_mat_trans (const short *x, short rows, short columns, short *r)

Arguments

x[rows*columns]	Pointer to input matrix.
rows	Number of rows in the input matrix. Must be a multiple of 4.
columns	Number of columns in the input matrix. Must be a multiple of 4.
r[columns*rows]	Pointer to output data vector of size rows*columns.

Description This function transposes the input matrix x[ ] and writes the result to matrix r[ ].

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_mat_trans(short *x, short rows, short columns, short
*r)
{
    short i,j;
    for(i=0; i<columns; i++)
        for(j=0; j<rows; j++)
            *(r+i*rows+j)=*(x+i+columns*j);
}
```

Special Requirements

- ❑ Rows and columns must be a multiple of 4.
- ❑ Matrices are assumed to have 16-bit elements.

Implementation Notes

- ❑ ***Bank Conflicts:*** No bank conflicts occur.
- ❑ ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- ❑ Data from four adjacent rows, spaced “columns” apart are read, and a local 4x4 transpose is performed in the register file. This leads to four double words, that are “rows” apart. These loads and stores can cause bank conflicts; hence, non-aligned loads and stores are used.

Benchmarks

Cycles	(2 * rows + 9) * columns/4 + 3
Codesize	224 bytes

4.7 Miscellaneous

DSP_bexp *Block Exponent Implementation*

Function short DSP_bexp(const int *x, short nx)

Arguments

x[nx]	Pointer to input vector of size nx. Must be double-word aligned.
nx	Number of elements in input vector. Must be multiple of 8.
return short	Return value is the maximum exponent that may be used in scaling.

Description Computes the exponents (number of extra sign bits) of all values in the input vector x[ ] and returns the minimum exponent. This will be useful in determining the maximum shift value that may be used in scaling a block of data.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
short DSP_bexp(const int *x, short nx)
{
    int       min_val =_norm(x[0]);
    short     n;
    int       i;
    for(i=1;i<nx;i++)
    {
        n =_norm(x[i]);  /* _norm(x) = number of */
                         /* redundant sign bits  */
        if(n<min_val) min_val=n;
    }
    return min_val;
}
```

Special Requirements

- ❑ nx must be a multiple of 8.
- ❑ The input vector x[ ] must be double-word aligned.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx/2 + 21
Codesize	216 bytes

DSP_blk_eswap16 *Endian-Swap a Block of 16-Bit Values*

Function void blk_eswap16(void * restrict x, void * restrict r, int nx)

Arguments

x [nx]	Source data. Must be double-word aligned.
r [nx]	Destination array. Must be double-word aligned.
nx	Number of 16-bit values to swap. Must be multiple of 8.

Description The data in the x[] array is endian swapped, meaning that the byte-order of the bytes within each half-word of the r[] array is reversed. This facilitates moving big-endian data to a little-endian system or vice-versa.

When the r pointer is non-NULL, the endian-swap occurs out-of-place, similar to a block move. When the r pointer is NULL, the endian-swap occurs in-place, allowing the swap to occur without using any additional storage.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_blk_eswap16(void *x, void *r, int  nx)
{
    int i;
    char *_x, *_r;

    if (r)
    {
        _x = (char *)x;
        _r = (char *)r;
    } else
    {
        _x = (char *)x;
        _r = (char *)r;
    }

    for (i = 0; i < nx; i++)
    {
        char t0, t1;
        t0 = _x[i*2 + 1];
        t1 = _x[i*2 + 0];
        _r[i*2 + 0] = t0;
        _r[i*2 + 1] = t1;
    }
}
```

Special Requirements

- Input and output arrays do not overlap, except when “r == NULL” so that the operation occurs in-place.
- The input array and output array are expected to be double-word aligned, and a multiple of 8 half-words must be processed.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx/8 + 18
Codesize	104 bytes

DSP_blk_eswap32 *Endian-Swap a Block of 32-Bit Values*

Function void blk_eswap32(void * restrict x, void * restrict r, int nx)

Arguments

x [nx]	Source data. Must be double-word aligned.
r [nx]	Destination array. Must be double-word aligned.
nx	Number of 32-bit values to swap. Must be multiple of 4.

Description The data in the x[] array is endian swapped, meaning that the byte-order of the bytes within each word of the r[] array is reversed. This facilitates moving big-endian data to a little-endian system or vice-versa.

When the r pointer is non-NULL, the endian-swap occurs out-of-place, similar to a block move. When the r pointer is NULL, the endian-swap occurs in-place, allowing the swap to occur without using any additional storage.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_blk_eswap32(void *x, void *r, int  nx)
{
    int i;
    char *_x, *_r;

    if (r)
    {
        _x = (char *)x;
        _r = (char *)r;
    } else
    {
        _x = (char *)x;
        _r = (char *)r;
    }

    for (i = 0; i < nx; i++)
    {
        char t0, t1, t2, t3;
        t0 = _x[i*4 + 3];
        t1 = _x[i*4 + 2];
```

```
            t2 = _x[i*4 + 1];
            t3 = _x[i*4 + 0];
            _r[i*4 + 0] = t0;
            _r[i*4 + 1] = t1;
            _r[i*4 + 2] = t2;
            _r[i*4 + 3] = t3;
        }
    }
```

Special Requirements

- Input and output arrays do not overlap, except where “r == NULL” so that the operation occurs in-place.
- The input array and output array are expected to be double-word aligned, and a multiple of 4 words must be processed.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx/4 + 20
Codesize	116 bytes

DSP_blk_eswap64 *Endian-Swap a Block of 64-Bit Values*

Function void blk_eswap64(void * restrict x, void * restrict r, int nx)

Arguments

x[nx]	Source data. Must be double-word aligned.
r[nx]	Destination array. Must be double-word aligned.
nx	Number of 64-bit values to swap. Must be multiple of 2.

Description The data in the x[] array is endian swapped, meaning that the byte-order of the bytes within each double-word of the r[] array is reversed. This facilitates moving big-endian data to a little-endian system or vice-versa.

When the r pointer is non-NULL, the endian-swap occurs out-of-place, similar to a block move. When the r pointer is NULL, the endian-swap occurs in-place, allowing the swap to occur without using any additional storage.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_blk_eswap64(void *x, void *r, int  nx)
{
    int i;
    char *_x, *_r;

    if (r)
    {
        _x = (char *)x;
        _r = (char *)r;
    } else
    {
        _x = (char *)x;
        _r = (char *)r;
    }

    for (i = 0; i < nx; i++)
    {
        char t0, t1, t2, t3, t4, t5, t6, t7;
        t0 = _x[i*8 + 7];
        t1 = _x[i*8 + 6];
```

```
            t2 = _x[i*8 + 5];
            t3 = _x[i*8 + 4];
            t4 = _x[i*8 + 3];
            t5 = _x[i*8 + 2];
            t6 = _x[i*8 + 1];
            t7 = _x[i*8 + 0];
            _r[i*8 + 0] = t0;
            _r[i*8 + 1] = t1;
            _r[i*8 + 2] = t2;
            _r[i*8 + 3] = t3;
            _r[i*8 + 4] = t4;
            _r[i*8 + 5] = t5;
            _r[i*8 + 6] = t6;
            _r[i*8 + 7] = t7;
        }
    }
```

Special Requirements

- Input and output arrays do not overlap, except when “r == NULL” so that the operation occurs in-place.
- The input array and output array are expected to be double-word aligned, and a multiple of 2 double-words must be processed.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	nx/2 + 20
Codesize	116 bytes

DSP_blk_move Block Move (Overlapping)

Function void DSP_blk_move(short * x, short * r, int nx)

Arguments

x [nx]	Block of data to be moved.
r [nx]	Destination of block of data.
nx	Number of elements in block. Must be multiple of 8 and ≥32.

Description This routine moves nx 16-bit elements from one memory location pointed to by x to another pointed to by r. The source and destination blocks can be overlapped.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_blk_move(short *x, short *r, int nx)
{
    int i;

    if( r < x )
    {
        for (I = 0; I < nx; i++)
            r[i] = x[i];
    } else
    {
        for (I = nx−1; I >= 0; i−−)
            r[i] = x[i];
    }
}
```

Special Requirements nx must be a multiple of 8 and ≥ 32.

Implementation Notes

- Twin input and output pointers are used.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is fully interruptible.

Benchmarks

Cycles	nx/4+18
Codesize	112 bytes

DSP_fltoq15 *Float to Q15 Conversion*

Function void DSP_fltoq15 (float *x, short *r, short nx)

Arguments

x[nx]	Pointer to floating-point input vector of size nx. x should contain the numbers normalized between [−1,1).
r[nx]	Pointer to output data vector of size nx containing the Q.15 equivalent of vector x.
nx	Length of input and output data vectors. Must be multiple of 2.

Description Convert the IEEE floating point numbers stored in vector x[ ] into Q.15 format numbers stored in vector r[ ]. Results are truncated toward zero. Values that exceed the size limit will be saturated to 0x7fff if value is positive and 0x8000 if value is negative. All values too small to be correctly represented will be truncated to 0.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void fltoq15(float x[], short r[], short nx)
{
    int i, a;

    for(i = 0; i < nx; i++)
    {
        a = 32768 * x[i];

        // saturate to 16−bit //
        if (a>32767)  a =  32767;
        if (a<−32768) a = −32768;

        r[i] = (short) a;
    }
}
```

Special Requirements nx must be a multiple of 2.

Implementation Notes

- Loop is unrolled twice.
- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

Cycles	3 * nx/2 + 14
Codesize	224 bytes

DSP_minerror — *Minimum Energy Error Search*

Function

int minerror (const short * restrict GSP0_TABLE, const short * restrict errCoefs, int * restrict max_index)

Arguments

GSP0_TABLE[9*256]	GSP0 terms array. Must be double-word aligned.
errCoefs[9]	Array of error coefficients.
max_index	Pointer to GSP0_TABLE[max_index] found.
return int	Maximum dot product result.

Algorithm

This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
int minerr
(
    const short *restrict GSP0_TABLE,
    const short *restrict errCoefs,
    int         *restrict max_index
)
{
    int val, maxVal = -50;
    int i, j;
    for (i = 0; i < GSP0_NUM; i++)
    {
        for (val = 0, j = 0; j < GSP0_TERMS; j++)
            val += GSP0_TABLE[i*GSP0_TERMS+j] * errCoefs[j];

        if (val > maxVal)
        {
            maxVal = val;
            *max_index = i*GSP0_TERMS;
        }
    }
    return (maxVal);
}
```

Special Requirements Array GSP0_TABLE[] must be double-word aligned.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- The load double-word instruction is used to simultaneously load four values in a single clock cycle.
- The inner loop is completely unrolled.
- The outer loop is 4 times unrolled.

Benchmarks

Cycles	256/4 * 9 + 17 = 593
Codesize	352 bytes

DSP_q15tofl *Q15 to Float Conversion*

Function	void DSP_q15tofl (short *x, float *r, int nx)	
Arguments	x[nx]	Pointer to Q.15 input vector of size nx.
	r[nx]	Pointer to floating-point output data vector of size nx containing the floating-point equivalent of vector x.
	nx	Length of input and output data vectors. Must be multiple of 2.

Description Converts the values stored in vector x[ ] in Q.15 format to IEEE floating point numbers in output vector r[ ].

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_q15tofl(short *x, float *r, int nx)
{
    int i;
    for (i=0;i<nx;i++)
        r[i] = (float) x[i] / 0x8000;
}
```

Special Requirements nx must be a multiple of 2.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- Loop is unrolled twice

Benchmarks		
	Cycles	2 * nx + 14
	Codesize	184 bytes

4.8 Obsolete Functions

4.8.1 FFT

DSP_bitrev_cplx *Complex Bit-Reverse*

NOTE: This function is provided for backward compatibility with the C62x DSPLIB. It has not been optimized for the C64x architecture. You are advised to use one of the newly added FFT functions which have been optimized for the C64x.

Function void DSP_bitrev_cplx (int *x, short *index, int nx)

Arguments

x[nx]	Pointer to complex input vector x of size nx
index[ ]	Array of size ~sqrt(nx) created by the routine digitrev_index (provided in the directory ‘support\fft’).
nx	Number of elements in vector x. nx must be a power of 2.

Description This function bit-reverses the position of elements in complex vector x. This function is used in conjunction with FFT routines to provide the correct format for the FFT input or output data. The bit-reversal of a bit-reversed order array yields a linear-order array.

Algorithm TI retains all rights, title and interest in this code and only authorizes the use of this code on TI TMS320 DSPs manufactured by TI. This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_bitrev_cplx (int *x, short *index, int nx)
{
    int       i;
    short     i0, i1, i2, i3;
    short     j0, j1, j2, j3;
    int       xi0, xi1, xi2, xi3;
    int       xj0, xj1, xj2, xj3;
    short     t;
    int       a, b, ia, ib, ibs;
    int       mask;
    int       nbits, nbot, ntop, ndiff, n2, halfn;
    short     *xs = (short *) x;
    nbits = 0;
    i = nx;
    while (i > 1){
       i = i >> 1;
       nbits++;}
    nbot   = nbits >> 1;
    ndiff  = nbits & 1;
    ntop   = nbot + ndiff;
    n2     = 1 << ntop;
    mask   = n2 - 1;
    halfn  = nx >> 1;
    for (i0 = 0; i0 < halfn; i0 += 2) {
       b   = i0 & mask;
       a   = i0 >> nbot;
       if (!b) ia   = index[a];
       ib  = index[b];
       ibs = ib << nbot;

       j0  = ibs + ia;
       t   = i0 < j0;
       xi0 = x[i0];
       xj0 = x[j0];
       if (t){x[i0] = xj0;
          x[j0] = xi0;}
       i1  = i0 + 1;
       j1  = j0 + halfn;
       xi1 = x[i1];
       xj1 = x[j1];
       x[i1] = xj1;
       x[j1] = xi1;
       i3  = i1 + halfn;
       j3  = j1 + 1;
       xi3 = x[i3];
       xj3 = x[j3];
```

```
                if (t){x[i3] = xj3;
                    x[j3] = xi3;}
            }
        }
```

Special Requirements

- nx must be a power of 2.
- The array index[] is generated by the routine bitrev_index provided in the directory ‘support\fft’.
- If nx ≤ 4K, you can use the char (8-bit) data type for the “index” variable. This requires changing the LDH when loading index values in the assembly routine to LDB. This further reduces the size of the Index Table by half.

Implementation Notes

- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

Benchmarks

The performance of this function has not yet been characterized on the C64x+

DSP_radix2 *Complex Forward FFT (radix 2)*

NOTE: This function is provided for backward compatibility with the C62x DSPLIB. It has not been optimized for the C64x architecture. You are advised to use one of the newly added FFT functions which have been optimized for the C64x.

Function void DSP_radix2 (int nx, short * restrict x, const short * restrict w)

Arguments

nx	Number of complex elements in vector x. Must be a power of 2 such that 4 ≤ nx ≤ 65536.
x[2*nx]	Pointer to input and output sequences. Size 2*nx elements.
w[nx]	Pointer to vector of FFT coefficients of size nx elements.

Description This routine is used to compute FFT of a complex sequence of size nx, a power of 2, with “decimation-in-frequency decomposition” method. The output is in bit-reversed order. Each complex value is with interleaved 16-bit real and imaginary parts.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_radix2 (short x[ ],short nx,short w[ ])
{
    short n1,n2,ie,ia,i,j,k,l;
    short xt,yt,c,s;

    n2 = nx;
    ie = 1;
    for (k=nx; k > 1; k = (k >> 1) ) {
        n1 = n2;
        n2 = n2>>1;
        ia = 0;
        for (j=0; j < n2; j++) {
            c = w[2*ia];
            s = w[2*ia+1];
            ia = ia + ie;
            for (i=j; i < nx; i += n1) {
                l = i + n2;
```

```
                    xt        = x[2*l] - x[2*i];
                    x[2*i]    = x[2*i] + x[2*l];
                    yt        = x[2*l+1] - x[2*i+1];
                    x[2*i+1]  = x[2*i+1] + x[2*l+1];
                    x[2*l]    = (c*xt + s*yt)>>15;
                    x[2*l+1]  = (c*yt - s*xt)>>15;
                }
            }
            ie = ie<<1;
        }
    }
```

Special Requirements

- $2 \le nx \le 32768$ (nx is a power of 2)
- Input x and coefficients w should be in different data sections or memory spaces to eliminate memory bank hits. If this is not possible, they should be aligned on different word boundaries to minimize memory bank hits.
- x data is stored in the order real[0], image[0], real[1], ...
- The FFT coefficients (twiddle factors) are generated using the program tw_radix2 provided in the directory ‘support\fft’.

Implementation Notes

- ***Bank Conflicts:*** See Benchmarks.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- Loads input x and coefficient w as words.
- Both loops j and i0 shown in the C code are placed in the INNERLOOP of the assembly code.

Benchmarks The performance of this function has not yet been characterized on the C64x+.

DSP_r4fft *Complex Forward FFT (radix 4)*

NOTE: This function is provided for backward compatibility with the C62x DSPLIB. It has not been optimized for the C64x architecture. You are advised to use one of the newly added FFT functions which have been optimized for the C64x.

Function void DSP_r4fft (int nx, short * restrict x, const short * restrict w)

Arguments

nx	Number of complex elements in vector x. Must be a power of 4 such that 4 ≤ nx ≤ 65536.
x[2*nx]	Pointer to input and output sequences. Size 2*nx elements.
w[nx]	Pointer to vector of FFT coefficients of size nx elements.

Description This routine is used to compute FFT of a complex sequence size nx, a power of 4, with “decimation-in-frequency decomposition” method. The output is in digit-reversed order. Each complex value is with interleaved 16-bit real and imaginary parts.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
void DSP_r4fft (int nx, short x[ ], short w[ ])
{
    int    n1, n2, ie, ia1, ia2, ia3, i0, i1, i2, i3,
           j, k;
    short  t, r1, r2, s1, s2, co1, co2, co3, si1,
           si2, si3;

    n2 = nx;
    ie = 1;
    for (k = nx; k > 1; k >>= 2) {
        n1 = n2;
        n2 >>= 2;
        ia1 = 0;
        for (j = 0; j < n2; j++) {
            ia2 = ia1 + ia1;
            ia3 = ia2 + ia1;
            co1 = w[ia1 * 2 + 1];
```

```
si1 = w[ia1 * 2];
co2 = w[ia2 * 2 + 1];
si2 = w[ia2 * 2];
co3 = w[ia3 * 2 + 1];
si3 = w[ia3 * 2];
ia1 = ia1 + ie;
for (i0 = j; i0 < nx; i0 += n1) {
   i1 = i0 + n2;
   i2 = i1 + n2;
   i3 = i2 + n2;
   r1 = x[2 * i0] + x[2 * i2];
   r2 = x[2 * i0] − x[2 * i2];
   t = x[2 * i1] + x[2 * i3];
   x[2 * i0] = r1 + t;
   r1 = r1 − t;
   s1 = x[2 * i0 + 1] + x[2 * i2 + 1];
   s2 = x[2 * i0 + 1] − x[2 * i2 + 1];
   t = x[2 * i1 + 1] + x[2 * i3 + 1];
   x[2 * i0 + 1] = s1 + t;
   s1 = s1 − t;
   x[2 * i2] = (r1 * co2 + s1 * si2) >>
   15;
   x[2 * i2 + 1] = (s1 * co2−r1 *
   si2)>>15;
   t = x[2 * i1 + 1] − x[2 * i3 + 1];
   r1 = r2 + t;
   r2 = r2 − t;
   t = x[2 * i1] − x[2 * i3];
   s1 = s2 − t;
   s2 = s2 + t;
   x[2 * i1] = (r1 * co1 + s1 * si1)
   >>15;
   x[2 * i1 + 1] = (s1 * co1−r1 *
   si1)>>15;
   x[2 * i3] = (r2 * co3 + s2 * si3)
```

```
                >>15;
                x[2 * i3 + 1] = (s2 * co3-r2 *
                si3)>>15;
            }
        }
        ie <<= 2;
    }
}
```

Special Requirements

- 4 ≤ nx ≤ 65536 (nx a power of 4)
- x is aligned on a 4*nx byte boundary for circular buffering
- Input x and coefficients w should be in different data sections or memory spaces to eliminate memory bank hits. If this is not possible, w should be aligned on an odd word boundary to minimize memory bank hits
- x data is stored in the order real[0], image[0], real[1], ...
- The FFT coefficients (twiddle factors) are generated using the program tw_r4fft provided in the directory ‘support\fft’.

Implementation Notes

- ***Bank Conflicts:*** See Benchmarks.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- Loads input x and coefficient w as words.
- Both loops j and i0 shown in the C code are placed in the INNERLOOP of the assembly code.

Benchmarks The performance of this function has not yet been characterized on the C64x+.

DSP_fft *Complex Forward FFT With Digital Reversal*

Function void DSP_fft (const short * restrict w, int nx, short * restrict x, short * restrict y)

Arguments

w[2*nx]	Pointer to vector of Q.15 FFT coefficients of size 2 * nx elements. Must be double-word aligned.
nx	Number of complex elements in vector x. Must be a power of 4 and 4 ≤ nx ≤ 65536.
x[2*nx]	Pointer to input sequence of size 2 * nx elements. Must be double-word aligned.
y[2*nx]	Pointer to output sequence of size 2 * nx elements. Must be double-word aligned.

Description This routine is used to compute an FFT of a complex sequence of size nx, a power of 4, with “decimation-in-frequency decomposition” method. The output is returned in a separate array y in normal order. This routine also performs digit reversal as a special last step. Each complex value is stored as interleaved 16-bit real and imaginary parts. The code uses a special ordering of FFT factors and memory accesses to improve performance in the presence of cache.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
/*------------------------------------------------------------------------*/
/* The following macro is used to obtain a digit reversed index, of a given */
/* number i, into j where the number of bits in ”i” is ”m”. For the natural */
/* form of C code, this is done by first interchanging every set of ”2 bit” */
/* pairs, followed by exchanging nibbles, followed by exchanging bytes, and */
/* finally halfwords. To give an example, consider the following number:    */
/*                                                                          */
/* N = FEDCBA9876543210, where each digit represents a bit, the following   */
/* steps illustrate the changes as the exchanges are performed:             */
/* M = DCFE98BA54761032 is the number after every ”2 bits” are exchanged.   */
/* O = 98BADCFE10325476 is the number after every nibble is exchanged.      */
/* P = 1032547698BADCFE is the number after every byte is exchanged.        */
/* Since only 16 digits were considered this represents the digit reversed  */
/* index. Since the numbers are represented as 32 bits, there is one more   */
/* step typically of exchanging the half words as well.                     */
/*------------------------------------------------------------------------*/
```

```
#include <stdio.h>
#include <stdlib.h>
#if 0
# define DIG_REV(i, m, j) ((j) = (_shfl(_rotl(_bitr(_deal(i)), 16)) >> (m)))
#else
# define DIG_REV(i, m, j)                                                  \
    do {                                                                   \
        unsigned _ = (i);                                                  \
        _ = ((_ & 0x33333333) <<  2) | ((_ & ~0x33333333) >>  2);         \
        _ = ((_ & 0x0F0F0F0F) <<  4) | ((_ & ~0x0F0F0F0F) >>  4);         \
        _ = ((_ & 0x00FF00FF) <<  8) | ((_ & ~0x00FF00FF) >>  8);         \
        _ = ((_ & 0x0000FFFF) << 16) | ((_ & ~0x0000FFFF) >> 16);         \
        (j) = _ >> (m);                                                    \
    } while (0)
#endif
void fft_cn
(
    const short *restrict w,
    int n,
    short       *restrict x,
    short       *restrict y
)
{
    int stride, i, j, k, t, s, m;

    short  xh0, xh1,  xh20,  xh21;
    short  xl0, xl1,  xl20,  xl21;
    short  xt0, yt0,  xt1,   yt1;
    short  xt2, yt2,  xt3,   yt3;
    /*----------------------------------------------------------------------*/
    /* Inform the compiler that the  input array "x", twiddle factor array  */
    /* "w" and output array "y" are double word aligned.  In addition, the   */
    /* number of points to be transformed is assumed to be greater than or  */
    /* equal to 16, and less than 32768.                                    */
    /*----------------------------------------------------------------------*/
    #ifndef NOASSUME
    _nassert((int)x % 8 == 0);
    _nassert((int)y % 8 == 0);
    _nassert((int)w % 8 == 0);
    _nassert(n >= 16);
    _nassert(n <  32768);
    #endif
    /* ---------------------------------------------------------------- */
    /*  Perform initial stages of FFT in place w/out digit reversal.    */
    /* ---------------------------------------------------------------- */
    #ifndef NOASSUME
    #pragma MUST_ITERATE(1,,1);
    #endif
    for (stride = n, t = 0; stride > 4; stride >>= 2)
    {
        /* ------------------------------------------------------------ */
        /*  Perform each of the butterflies for this particular stride. */
        /* ------------------------------------------------------------ */
        s = stride >> 2;
        /*--------------------------------------------------------------*/
        /* stride represents the seperation between the inputs of the radix */
        /* 4 butterfly. The C code breaks the FFT, into two cases, one when */
        /* the stride between the elements is greater than 4, other when    */
        /* the stride is less than 4. Since stride is greater than 16, it   */
        /* can be guaranteed that ”s” is greater than or equal to 4.         */
        /* In addition, it can also be shown that the loop that shares this  */
        /* stride will iterate at least once.  The number of times this     */
        /* loop iterates depends on how many butterflies in this stage      */
        /* share a twiddle factor.                                          */
        /*--------------------------------------------------------------*/

        #ifndef NOASSUME
        _nassert(stride >= 16);
        _nassert(s      >=  4);
        #pragma MUST_ITERATE(1,,1);
        #endif
        for (i = 0; i < n; i += stride)
```

```
{
    #ifndef NOASSUME
    _nassert(i % 4 == 0);
    _nassert(s     >= 4);
    #pragma MUST_ITERATE(2,,2);
    #endif
    for (j = 0; j < s; j += 2)
    {
        for (k = 0; k < 2; k++)
        {
            short           w1c, w1s, w2c, w2s, w3c, w3s;
            short x0r, x0i, x1r, x1i, x2r, x2i, x3r, x3i;
            short y0r, y0i, y1r, y1i, y2r, y2i, y3r, y3i;
            /* ---------------------------------------------------- */
            /*  Read the four samples that are the input to this     */
            /*  particular butterfly.                                */
            /* ---------------------------------------------------- */
            x0r = x[2*(i+j+k      ) + 0]; x0i = x[2*(i+j+k      ) + 1];
            x1r = x[2*(i+j+k +   s) + 0]; x1i = x[2*(i+j+k +   s) + 1];
            x2r = x[2*(i+j+k + 2*s) + 0]; x2i = x[2*(i+j+k + 2*s) + 1];
            x3r = x[2*(i+j+k + 3*s) + 0]; x3i = x[2*(i+j+k + 3*s) + 1];
            /* ---------------------------------------------------- */
            /*  Read the six twiddle factors that are needed for 3   */
            /*  of the four outputs. (The first output has no mpys.) */
            /* -----------------------------------------------------*/
            w1s = w[t + 2*k + 6*j + 0];    w1c = w[t + 2*k + 6*j + 1];
            w2s = w[t + 2*k + 6*j + 4];    w2c = w[t + 2*k + 6*j + 5];
            w3s = w[t + 2*k + 6*j + 8];    w3c = w[t + 2*k + 6*j + 9];
            /* ---------------------------------------------------- */
            /*  Calculate the four outputs, remembering that radix4  */
            /*  FFT accepts 4 inputs and produces 4 outputs. If we   */
            /*  imagine the inputs as being complex, and look at the */
            /*  first stage as an example:                           */
            /*                                                       */
            /*  Four inputs are x(n) x(n+N/4) x(n+N/2) x(n+3N/4)     */
            /*  In general the four inputs can be generalized using  */
            /*  the stride between the elements as follows:          */
            /*  x(n), x(n + s), x(n + 2*s), x(n + 3*s).              */
            /*                                                       */
            /*  These four inputs are used to calculate four outputs */
            /*  as shown below:                                      */
            /*                                                       */
            /* X(4k)  = x(n) + x(n + N/4) + x(n + N/2) + x(n + 3N/4) */
            /* X(4k+1)= x(n) -jx(n + N/4) - x(n + N/2) +jx(n + 3N/4) */
            /* X(4k+2)= x(n) - x(n +N/4)  + x(N + N/2) - x(n + 3N/4) */
            /* X(4k+3)= x(n) +jx(n + N/4) - x(n + N/2) -jx(n + 3N/4) */
            /*                                                       */
            /* These four partial results can be re-written to show  */
            /* the underlying DIF structure similar to DSP_radix2 as */
            /* follows:                                              */
            /*                                                       */
            /* X(4k)  = (x(n)+x(n + N/2)) + (x(n+N/4)+ x(n + 3N/4))  */
            /* X(4k+1)= (x(n)-x(n + N/2)) -j(x(n+N/4) - x(n + 3N/4)) */
            /* x(4k+2)= (x(n)+x(n + N/2)) - (x(n+N/4)+ x(n + 3N/4))  */
            /* X(4k+3)= (x(n)-x(n + N/2)) +j(x(n+N/4) - x(n + 3N/4)) */
            /*                                                       */
            /* which leads to the real and imaginary values as foll: */
            /*                                                       */
            /* y0r = x0r + x2r +  x1r +  x3r    =  xh0 + xh20        */
            /* y0i = x0i + x2i +  x1i +  x3i    =  xh1 + xh21        */
            /* y1r = x0r - x2r + (x1i -  x3i)   =  xl0 + xl21        */
            /* y1i = x0i - x2i - (x1r -  x3r)   =  xl1 - xl20        */
            /* y2r = x0r + x2r - (x1r +  x3r)   =  xh0 - xh20        */
            /* y2i = x0i + x2i - (x1i +  x3i    =  xh1 - xh21        */
            /* y3r = x0r - x2r - (x1i -  x3i)   =  xl0 - xl21        */
            /* y3i = x0i - x2i + (x1r -  x3r)   =  xl1 + xl20        */
            /* ----------------------------------------------------- */
            xh0  = x0r   +   x2r;
            xh1  = x0i   +   x2i;
            xh20 = x1r   +   x3r;
            xh21 = x1i   +   x3i;
            xl0  = x0r   -   x2r;
```

```
                xl1  = x0i   -   x2i;
                xl20 = x1r   -   x3r;
                xl21 = x1i   -   x3i;
                xt0  =  xh0  +   xh20;
                yt0  =  xh1  +   xh21;
                xt1  =  xl0  +   xl21;
                yt1  =  xl1  -   xl20;
                xt2  =  xh0  -   xh20;
                yt2  =  xh1  -   xh21;
                xt3  =  xl0  -   xl21;
                yt3  =  xl1  +   xl20;
                /*-----------------------------------------------------*/
                /* Perform twiddle factor multiplies of three terms,top  */
                /* term does not have any multiplies. Note the twiddle   */
                /* factors for a normal FFT are C + j (-S). Since the    */
                /* factors that are stored are C + j S, this is          */
                /* corrected for in the multiplies.                      */
                /*                                                       */
                /* Y1 = (xt1 + jyt1) (c + js) = (xc + ys) + (yc -xs)     */
                /*-----------------------------------------------------*/

                y0r = xt0;
                y0i = yt0;
                y1r = (xt1 * w1c +  yt1 * w1s) >> 15;
                y1i = (yt1 * w1c -  xt1 * w1s) >> 15;
                y2r = (xt2 * w2c +  yt2 * w2s) >> 15;
                y2i = (yt2 * w2c -  xt2 * w2s) >> 15;
                y3r = (xt3 * w3c +  yt3 * w3s) >> 15;
                y3i = (yt3 * w3c -  xt3 * w3s) >> 15;

                /* --------------------------------------------------- */
                /*  Store the final results back to the input array.    */
                /* --------------------------------------------------- */

                x[2*(i+j+k      ) + 0] = y0r; x[2*(i+j+k      ) + 1] = y0i;
                x[2*(i+j+k +   s) + 0] = y1r; x[2*(i+j+k +   s) + 1] = y1i;
                x[2*(i+j+k + 2*s) + 0] = y2r; x[2*(i+j+k + 2*s) + 1] = y2i;
                x[2*(i+j+k + 3*s) + 0] = y3r; x[2*(i+j+k + 3*s) + 1] = y3i;
            }
        }
    }
```

```
        /* -------------------------------------------------------------- */
        /*  Offset to next subtable of twiddle factors. With each iteration */
        /*  of the above block, six twiddle factors get read, s times,      */
        /*  hence the offset into the twiddle factor array is advanced by   */
        /*  this amount.                                                    */
        /* -------------------------------------------------------------- */
        t += 6 * s;
    }
    /* ---------------------------------------------------------------------- */
    /*  Get the magnitude of ”n”, so we know how many digits to reverse.    */
    /* ---------------------------------------------------------------------- */
    for (i = 31, m = 1; (n & (1 << i)) == 0; i--, m++) ;
    /* ---------------------------------------------------------------------- */
    /*  Perform final stage with digit reversal.                            */
    /* ---------------------------------------------------------------------- */
    s = n >> 2;
    /*------------------------------------------------------------------------*/
    /* One of the nice features, of this last stage are that, no multiplies */
    /* are required. In addition, the data always strides by a fixed amount  */
    /* namely 8 elements. Since the data is stored as interleaved pairs, of */
    /* real and imaginary data, the first eight elements contain the data   */
    /* for the first four complex inputs. These are the inputs to the first */
    /* radix4 butterfly.                                                    */
    /*------------------------------------------------------------------------*/
    #ifndef NOASSUME
    #pragma MUST_ITERATE(4,,4);
    #endif
    for (i = 0; i < n; i += 4)
    {
        short x0r, x0i, x1r, x1i, x2r, x2i, x3r, x3i;
        short y0r, y0i, y1r, y1i, y2r, y2i, y3r, y3i;

        /* -------------------------------------------------------------- */
        /*  Read the four samples that are the input to this butterfly.     */
        /* -------------------------------------------------------------- */
        x0r = x[2*(i + 0) + 0];     x0i = x[2*(i + 0) + 1];
        x1r = x[2*(i + 1) + 0];     x1i = x[2*(i + 1) + 1];
        x2r = x[2*(i + 2) + 0];     x2i = x[2*(i + 2) + 1];
        x3r = x[2*(i + 3) + 0];     x3i = x[2*(i + 3) + 1];
        /* ---------------------------------------------------------------- */
        /*  Calculate the final FFT result from this butterfly.             */
        /* ---------------------------------------------------------------- */
        y0r  = (x0r + x2r) + (x1r + x3r);
        y0i  = (x0i + x2i) + (x1i + x3i);
        y1r  = (x0r - x2r) + (x1i - x3i);
        y1i  = (x0i - x2i) - (x1r - x3r);
        y2r  = (x0r + x2r) - (x1r + x3r);
        y2i  = (x0i + x2i) - (x1i + x3i);
        y3r  = (x0r - x2r) - (x1i - x3i);
        y3i  = (x0i - x2i) + (x1r - x3r);
        /* ---------------------------------------------------------------- */
        /*  Digit reverse our address to convert the digit-reversed input   */
        /*  into a linearized output order.  This actually results in a     */
        /*  digit-reversed store pattern since we're loading linearly, but  */
        /*  the end result is that the FFT bins are in linear order.        */
        /* ---------------------------------------------------------------- */
        DIG_REV(i, m, j); /* Note:  Result is assigned to 'j' by the macro. */
        /* ---------------------------------------------------------------- */
        /*  Store out the final FFT results.                                */
        /* ---------------------------------------------------------------- */
        y[2*(j +   0) + 0] = y0r;   y[2*(j +   0) + 1] = y0i;
        y[2*(j +   s) + 0] = y1r;   y[2*(j +   s) + 1] = y1i;
        y[2*(j + 2*s) + 0] = y2r;   y[2*(j + 2*s) + 1] = y2i;
        y[2*(j + 3*s) + 0] = y3r;   y[2*(j + 3*s) + 1] = y3i;
    }
}
```

Special Requirements

- In-place computation is *not* allowed.
- nx must be a power of 4 and $4 \le nx \le 65536$.
- Input x[ ] and output y[ ] are stored on double-word aligned boundaries.
- Input data x[ ] is stored in the order real0, img0, real1, img1, ...
- The FFT coefficients (twiddle factors) must be double-word aligned and are generated using the program tw_fft16x16 provided in the directory ‘support\fft’.

Implementation Notes

- ***Bank Conflicts:*** No bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.
- Loads input x[ ] and coefficient w[ ] as double words.
- Both loops j and i0 shown in the C code are placed in the inner loop of the assembly code.

Benchmarks

Cycles	$1.25 * nx * \log_4(nx) - 0.5 * nx + 23 * \log_4(nx) - 1$
Codesize	984 bytes

DSP_fft16x16t *Complex Forward Mixed Radix 16- x 16-Bit FFT With Truncation*

Function void DSP_fft16x16t(const short * restrict w, int nx, short * restrict x, short * restrict y)

Arguments

w[2*nx]	Pointer to complex Q.15 FFT coefficients.
nx	Length of FFT in complex samples. Must be power of 2 or 4 , and 16 ≤ nx ≤ 32768.
x[2*nx]	Pointer to complex 16-bit data input.
y[2*nx]	Pointer to complex 16-bit data output.

Description This routine computes a complex forward mixed radix FFT with truncation and digit reversal. Input data x[ ], output data y[ ], and coefficients w[ ] are 16-bit. The output is returned in the separate array y[ ] in normal order. Each complex value is stored with interleaved real and imaginary parts. The code uses a special ordering of FFT coefficients (also called twiddle factors) and memory accesses to improve performance in the presence of cache.

Algorithm This is the C equivalent of the assembly code without restrictions. Note that the assembly code is hand optimized and restrictions may apply.

```
/*-----------------------------------------------------------------------*/
/* The following macro is used to obtain a digit reversed index, of a given */
/* number i, into j where the number of bits in ”i” is ”m”. For the natural */
/* form of C code, this is done by first interchanging every set of ”2 bit” */
/* pairs, followed by exchanging nibbles, followed by exchanging bytes, and */
/* finally halfwords. To give an example, consider the following number:    */
/*                                                                          */
/* N = FEDCBA9876543210, where each digit represents a bit, the following   */
/* steps illustrate the changes as the exchanges are performed:             */
/* M = DCFE98BA54761032 is the number after every ”2 bits” are exchanged.   */
/* O = 98BADCFE10325476 is the number after every nibble is exchanged.      */
/* P = 1032547698BADCFE is the number after every byte is exchanged.        */
/* Since only 16 digits were considered this represents the digit reversed  */
/* index. Since the numbers are represented as 32 bits, there is one more   */
/* step typically of exchanging the half words as well.                     */
/*-----------------------------------------------------------------------*/
#if TMS320C6X
```

```
# define DIG_REV(i, m, j) ((j) = (_shfl(_rotl(_bitr(_deal(i)), 16)) >> (m)))
#else
# define DIG_REV(i, m, j)                                                       \
    do {                                                                        \
        unsigned _ = (i);                                                       \
        _ = ((_ & 0x33333333) <<  2) | ((_ & ~0x33333333) >>  2);           \
        _ = ((_ & 0x0F0F0F0F) <<  4) | ((_ & ~0x0F0F0F0F) >>  4);           \
        _ = ((_ & 0x00FF00FF) <<  8) | ((_ & ~0x00FF00FF) >>  8);           \
        _ = ((_ & 0x0000FFFF) << 16) | ((_ & ~0x0000FFFF) >> 16);           \
        (j) = _ >> (m);                                                         \
    } while (0)
#endif

void DSP_fft16x16t_cn(const short *restrict ptr_w, int  npoints, short * ptr_x,
                  short * ptr_y)
{
    int   i, j, l1, l2, h2, predj, tw_offset, stride, fft_jmp;
    short xt0_0, yt0_0, xt1_0, yt1_0, xt2_0, yt2_0;
    short xt0_1, yt0_1, xt1_1, yt1_1, xt2_1, yt2_1;
    short xh0_0, xh1_0, xh20_0, xh21_0, xl0_0, xl1_0, xl20_0, xl21_0;
    short xh0_1, xh1_1, xh20_1, xh21_1, xl0_1, xl1_1, xl20_1, xl21_1;
    short x_0, x_1, x_2, x_3, x_l1_0, x_l1_1, x_l1_2, x_l1_3, x_l2_0, x_l2_1;
    short xh0_2, xh1_2, xl0_2, xl1_2, xh0_3, xh1_3, xl0_3, xl1_3;
    short x_4, x_5, x_6, x_7, x_l2_2, x_l2_3, x_h2_0, x_h2_1, x_h2_2, x_h2_3;
    short x_8, x_9, x_a, x_b, x_c, x_d, x_e, x_f;
    short si10, si20, si30, co10, co20, co30;
    short si11, si21, si31, co11, co21, co31;
    short * x, * x2, * x0;
    short * y0, * y1, * y2, *y3;
    short n00, n10, n20, n30, n01, n11, n21, n31;
    short n02, n12, n22, n32, n03, n13, n23, n33;
    short y0r, y0i, y4r, y4i;
    int   n0, j0;
    int   radix,  m;
    int   norm;
    const short *w;
```

```
/*-------------------------------------------------------------------*/
/* Determine the magnitude od the number of points to be transformed.  */
/* Check whether we can use a radix4 decomposition or a mixed radix    */
/* transformation, by determining modulo 2.                            */
/*-------------------------------------------------------------------*/
for (i = 31, m = 1; (npoints & (1 << i)) == 0; i--, m++) ;
radix     =   m & 1 ? 2 :  4;
norm      =   m - 2;
/*--------------------------------------------------------------------*/
/* The stride is quartered with every iteration of the outer loop. It   */
/* denotes the seperation between any two adjacent inputs to the butter */
/* -fly. This should start out at N/4, hence stride is initially set to */
/* N. For every stride, 6*stride twiddle factors are accessed. The      */
/* ”tw_offset” is the offset within the current twiddle factor sub-     */
/* table. This is set to zero, at the start of the code and is used to  */
/* obtain the appropriate sub-table twiddle pointer by offsetting it     */
/* with the base pointer ”ptr_w”.                                       */
/*--------------------------------------------------------------------*/
stride     =   npoints;
tw_offset  =   0;
fft_jmp    =   6 * stride;
while (stride > radix)
{
    /*-----------------------------------------------------------------*/
    /* At the start of every iteration of the outer loop, ”j” is set   */
    /* to zero, as ”w” is pointing to the correct location within the  */
    /* twiddle factor array. For every iteration of the inner loop     */
    /* 6 * stride twiddle factors are accessed. For eg,                */
    /*                                                                 */
    /* #Iteration of outer loop  # twiddle factors    #times cycled    */
    /*  1                          6 N/4               1               */
    /*  2                          6 N/16              4               */
    /*  ...                                                            */
    /*-----------------------------------------------------------------*/
    j         = 0;
    fft_jmp >>= 2;
```

```
/*-----------------------------------------------------------------*/
/* Set up offsets to access ”N/4”, ”N/2”, ”3N/4” complex point or  */
/* ”N/2”, ”N”, ”3N/2” half word                                    */
/*-----------------------------------------------------------------*/
h2 = stride>>1;
l1 = stride;
l2 = stride + (stride >> 1);
/*-----------------------------------------------------------------*/
/*  Reset ”x” to point to the start of the input data array.       */
/* ”tw_offset” starts off at 0, and increments by ”6 * stride”     */
/*  The stride quarters with every iteration of the outer loop     */
/*-----------------------------------------------------------------*/
x = ptr_x;
w = ptr_w + tw_offset;
tw_offset += fft_jmp;
stride >>=   2;
/*----------------------------------------------------------------*/
/* The following loop iterates through the different butterflies, */
/* within a given stage. Recall that there are logN to base 4     */
/* stages. Certain butterflies share the twiddle factors. These   */
/* are grouped together. On the very first stage there are no     */
/* butterflies that share the twiddle factor, all N/4 butter-     */
/* flies have different factors. On the next stage two sets of    */
/* N/8 butterflies share the same twiddle factor. Hence, after     */
/* half the butterflies are performed, j the index into the       */
/* factor array resets to 0, and the twiddle factors are reused.  */
/* When this happens, the data pointer ’x’ is incremented by the  */
/* fft_jmp amount. In addition, the following code is unrolled to  */
/* perform ”2” radix4 butterflies in parallel.                    */
/*----------------------------------------------------------------*/
for (i = 0; i < npoints; i += 8)
{
    /*------------------------------------------------------------*/
    /* Read the first 12 twiddle factors, six of which are used   */
    /* for one radix 4 butterfly and six of which are used for    */
    /* next one.                                                  */
```

```
/*------------------------------------------------------------*/
co10 = w[j+1];    si10 = w[j+0];
co11 = w[j+3];    si11 = w[j+2];
co20 = w[j+5];    si20 = w[j+4];
co21 = w[j+7];    si21 = w[j+6];
co30 = w[j+9];    si30 = w[j+8];
co31 = w[j+11];   si31 = w[j+10];
/*------------------------------------------------------------*/
/* Read in the first complex input for the butterflies.       */
/* 1st complex input to 1st butterfly: x[0] + jx[1]           */
/* 1st complex input to 2nd butterfly: x[2] + jx[3]           */
/*------------------------------------------------------------*/
x_0 = x[0];       x_1 = x[1];
x_2 = x[2];       x_3 = x[3];
/*------------------------------------------------------------*/
/* Read in the complex inputs for the butterflies. Each of the*/
/* successive complex inputs of the butterfly are seperated   */
/* by a fixed amount known as stride. The stride starts out   */
/* at N/4, and quarters with every stage.                     */
/*------------------------------------------------------------*/
x_l1_0 = x[l1  ]; x_l1_1 = x[l1+1];
x_l1_2 = x[l1+2]; x_l1_3 = x[l1+3];
x_l2_0 = x[l2  ]; x_l2_1 = x[l2+1];
x_l2_2 = x[l2+2]; x_l2_3 = x[l2+3];
x_h2_0 = x[h2  ]; x_h2_1 = x[h2+1];
x_h2_2 = x[h2+2]; x_h2_3 = x[h2+3];
/*-----------------------------------------------------------*/
/* Two butterflies are evaluated in parallel. The following  */
/* results will be shown for one butterfly only, although    */
/* both are being evaluated in parallel.                     */
/*                                                           */
/* Perform DSP_radix2 style DIF butterflies.                 */
/*-----------------------------------------------------------*/
xh0_0  = x_0    + x_l1_0;     xh1_0  = x_1    + x_l1_1;
xh0_1  = x_2    + x_l1_2;     xh1_1  = x_3    + x_l1_3;
xl0_0  = x_0    - x_l1_0;     xl1_0  = x_1    - x_l1_1;
```

```
xl0_1  = x_2    - x_l1_2;     xl1_1  = x_3    - x_l1_3;
xh20_0 = x_h2_0 + x_l2_0;     xh21_0 = x_h2_1 + x_l2_1;
xh20_1 = x_h2_2 + x_l2_2;     xh21_1 = x_h2_3 + x_l2_3;
xl20_0 = x_h2_0 - x_l2_0;     xl21_0 = x_h2_1 - x_l2_1;
xl20_1 = x_h2_2 - x_l2_2;     xl21_1 = x_h2_3 - x_l2_3;
/*-----------------------------------------------------------*/
/* Derive output pointers using the input pointer ”x”        */
/*-----------------------------------------------------------*/
x0 = x;
x2 = x0;
/*-----------------------------------------------------------*/
/* When the twiddle factors are not to be reused, j is      */
/* incremented by 12, to reflect the fact that 12 half words */
/* are consumed in every iteration. The input data pointer   */
/* increments by 4. Note that within a stage, the stride     */
/* does not change and hence the offsets for the other three */
/* legs, 0, h2, l1, l2.                                      */
/*-----------------------------------------------------------*/
j += 12;
x += 4;
predj = (j - fft_jmp);
if (!predj) x += fft_jmp;
if (!predj) j = 0;
/*----------------------------------------------------------*/
/* These four partial results can be re-written to show     */
/* the underlying DIF structure similar to DSP_radix2 as    */
/* follows:                                                 */
/*                                                          */
/* X(4k)  = (x(n)+x(n + N/2)) + (x(n+N/4)+ x(n + 3N/4))     */
/* X(4k+1)= (x(n)-x(n + N/2)) -j(x(n+N/4) - x(n + 3N/4))    */
/* x(4k+2)= (x(n)+x(n + N/2)) - (x(n+N/4)+ x(n + 3N/4))     */
/* X(4k+3)= (x(n)-x(n + N/2)) +j(x(n+N/4) - x(n + 3N/4))    */
/*                                                          */
/* which leads to the real and imaginary values as foll:    */
/*                                                          */
/* y0r = x0r + x2r +  x1r +  x3r    =  xh0 + xh20           */
/* y0i = x0i + x2i +  x1i +  x3i    =  xh1 + xh21           */
/* y1r = x0r − x2r + (x1i −  x3i)   =  xl0 + xl21           */
/* y1i = x0i − x2i − (x1r −  x3r)   =  xl1 − xl20           */
/* y2r = x0r + x2r − (x1r +  x3r)   =  xh0 − xh20           */
/* y2i = x0i + x2i − (x1i +  x3i    =  xh1 − xh21           */
/* y3r = x0r − x2r − (x1i −  x3i)   =  xl0 − xl21           */
/* y3i = x0i − x2i + (x1r −  x3r)   =  xl1 + xl20           */
/* ---------------------------------------------------------*/
y0r   = xh0_0 + xh20_0; y0i   = xh1_0 + xh21_0;
y4r   = xh0_1 + xh20_1; y4i   = xh1_1 + xh21_1;
xt0_0 = xh0_0 − xh20_0;  yt0_0 = xh1_0 − xh21_0;
xt0_1 = xh0_1 − xh20_1;  yt0_1 = xh1_1 − xh21_1;
xt1_0 = xl0_0 + xl21_0;  yt2_0 = xl1_0 + xl20_0;
xt2_0 = xl0_0 − xl21_0;  yt1_0 = xl1_0 − xl20_0;
xt1_1 = xl0_1 + xl21_1;  yt2_1 = xl1_1 + xl20_1;
xt2_1 = xl0_1 − xl21_1;  yt1_1 = xl1_1 − xl20_1;
/*--------------------------------------------------------*/
/* Store out first output, of the four outputs of a radix4 */
/* butterfly. Since two such radix4 butterflies are per−   */
/* formed in parallel, there are 2 such 1st outputs.       */
/*--------------------------------------------------------*/
x2[0] = y0r;             x2[1] = y0i;
x2[2] = y4r;             x2[3] = y4i;
/*--------------------------------------------------------*/
/* Perform twiddle factor multiplies of three terms,top    */
/* term does not have any multiplies. Note the twiddle     */
/* factors for a normal FFT are C + j (−S). Since the      */
/* factors that are stored are C + j S, this is            */
/* corrected for in the multiplies.                        */
/*                                                         */
/* Y1 = (xt1 + jyt1) (c + js) = (xc + ys) + (yc −xs)       */
/* Perform the multiplies using 16 by 32 multiply macro    */
/* defined. This treats the twiddle factor as 16 bits      */
/* and incoming data as 32 bits.                           */
/*--------------------------------------------------------*/
x2[h2  ] = (si10 * yt1_0 + co10 * xt1_0) >> 15;
```

```
            x2[h2+1] = (co10 * yt1_0 - si10 * xt1_0) >> 15;
            x2[h2+2] = (si11 * yt1_1 + co11 * xt1_1) >> 15;
            x2[h2+3] = (co11 * yt1_1 - si11 * xt1_1) >> 15;
            x2[l1  ] = (si20 * yt0_0 + co20 * xt0_0) >> 15;
            x2[l1+1] = (co20 * yt0_0 - si20 * xt0_0) >> 15;
            x2[l1+2] = (si21 * yt0_1 + co21 * xt0_1) >> 15;
            x2[l1+3] = (co21 * yt0_1 - si21 * xt0_1) >> 15;
            x2[l2  ] = (si30 * yt2_0 + co30 * xt2_0) >> 15;
            x2[l2+1] = (co30 * yt2_0 - si30 * xt2_0) >> 15;
            x2[l2+2] = (si31 * yt2_1 + co31 * xt2_1) >> 15;
            x2[l2+3] = (co31 * yt2_1 - si31 * xt2_1) >> 15;
        }
    }
    /*-----------------------------------------------------------------*/
    /* The following code performs either a standard radix4 pass or a  */
    /* DSP_radix2 pass. Two pointers are used to access the input data.*/
    /* The input data is read ”N/4” complex samples apart or ”N/2”     */
    /* words apart using pointers ”x0” and ”x2”. This produces out-    */
    /* puts that are 0, N/4, N/2, 3N/4 for a radix4 FFT, and 0, N/8    */
    /* N/2, 3N/8 for radix 2.                                          */
    /*-----------------------------------------------------------------*/
    y0 = ptr_y;
    y2 = ptr_y + (int) npoints;
    x0 = ptr_x;
    x2 = ptr_x + (int) (npoints >> 1);
    if (radix == 2)
    {
      /*-----------------------------------------------------------------*/
      /* The pointers are set at the following locations which are half */
      /* the offsets of a radix4 FFT.                                    */
      /*-----------------------------------------------------------------*/
      y1 = y0 + (int) (npoints >> 2);
      y3 = y2 + (int) (npoints >> 2);
      l1 = norm + 1;
      j0 = 8;
      n0 = npoints>>1;
```

```
}
else
{
  y1 = y0 + (int) (npoints >> 1);
  y3 = y2 + (int) (npoints >> 1);
  l1 = norm + 2;
  j0 = 4;
  n0 = npoints >> 2;
}
/*-------------------------------------------------------------------*/
/* The following code reads data indentically for either a radix 4    */
/* or a radix 2 style decomposition. It writes out at different       */
/* locations though. It checks if either half the points, or a        */
/* quarter of the complex points have been exhausted to jump to       */
/* pervent double reversal.                                           */
/*-------------------------------------------------------------------*/
j = 0;
for (i = 0; i < npoints; i += 8)
{
    /*---------------------------------------------------------------*/
    /* Digit reverse the index starting from 0. The increment to "j"  */
    /* is either by 4, or 8.                                          */
    /*---------------------------------------------------------------*/
    DIG_REV(j, l1, h2);
    /*---------------------------------------------------------------*/
    /* Read in the input data, from the first eight locations. These  */
    /* are transformed either as a radix4 or as a radix 2.            */
    /*---------------------------------------------------------------*/
    x_0 = x0[0];             x_1 = x0[1];
    x_2 = x0[2];             x_3 = x0[3];
    x_4 = x0[4];             x_5 = x0[5];
    x_6 = x0[6];             x_7 = x0[7];
    x0 += 8;
    xh0_0 = x_0 + x_4;       xh1_0 = x_1 + x_5;
    xl0_0 = x_0 - x_4;       xl1_0 = x_1 - x_5;
    xh0_1 = x_2 + x_6;       xh1_1 = x_3 + x_7;
```

```
        xl0_1 = x_2 - x_6;       xl1_1 = x_3 - x_7;
        n00 = xh0_0 + xh0_1; n01 = xh1_0 + xh1_1;
        n10 = xl0_0 + xl1_1; n11 = xl1_0 - xl0_1;
        n20 = xh0_0 - xh0_1; n21 = xh1_0 - xh1_1;
        n30 = xl0_0 - xl1_1; n31 = xl1_0 + xl0_1;
        if (radix == 2)
        {
             /*------------------------------------------------------------*/
             /* Perform DSP_radix2 style decomposition.                    */
             /*------------------------------------------------------------*/
             n00 = x_0 + x_2;     n01 = x_1 + x_3;
             n20 = x_0 - x_2;     n21 = x_1 - x_3;
             n10 = x_4 + x_6;     n11 = x_5 + x_7;
             n30 = x_4 - x_6;     n31 = x_5 - x_7;
        }
        y0[2*h2] = n00;           y0[2*h2 + 1] = n01;
        y1[2*h2] = n10;           y1[2*h2 + 1] = n11;
        y2[2*h2] = n20;           y2[2*h2 + 1] = n21;
        y3[2*h2] = n30;           y3[2*h2 + 1] = n31;
        /*------------------------------------------------------------*/
        /* Read in the next eight inputs, and perform radix4 or DSP_radix2*/
        /* decomposition.                                             */
        /*------------------------------------------------------------*/
        x_8 = x2[0];              x_9 = x2[1];
        x_a = x2[2];              x_b = x2[3];
        x_c = x2[4];              x_d = x2[5];
        x_e = x2[6];              x_f = x2[7];
        x2 += 8;
        xh0_2 = x_8 + x_c;        xh1_2  = x_9 + x_d;
        xl0_2 = x_8 - x_c;        xl1_2  = x_9 - x_d;
        xh0_3 = x_a + x_e;        xh1_3 = x_b + x_f;
        xl0_3 = x_a - x_e;        xl1_3 = x_b - x_f;
        n02 = xh0_2 + xh0_3;      n03 = xh1_2 + xh1_3;
        n12 = xl0_2 + xl1_3;      n13 = xl1_2 - xl0_3;
        n22 = xh0_2 - xh0_3;      n23 = xh1_2 - xh1_3;
        n32 = xl0_2 - xl1_3;      n33 = xl1_2 + xl0_3;
        if (radix == 2)
        {
          n02 = x_8 + x_a;        n03 = x_9 + x_b;
          n22 = x_8 - x_a;        n23 = x_9 - x_b;
          n12 = x_c + x_e;        n13 = x_d + x_f;
          n32 = x_c - x_e;        n33 = x_d - x_f;
        }
        /*-----------------------------------------------------------------*/
        /* Points that are read from succesive locations map to y, y[N/4]  */
        /* y[N/2], y[3N/4] in a radix4 scheme, y, y[N/8], y[N/2],y[5N/8]   */
        /*-----------------------------------------------------------------*/
        y0[2*h2+2] = n02;         y0[2*h2+3] = n03;
        y1[2*h2+2] = n12;         y1[2*h2+3] = n13;
        y2[2*h2+2] = n22;         y2[2*h2+3] = n23;
        y3[2*h2+2] = n32;         y3[2*h2+3] = n33;
        /*-----------------------------------------------------------------*/
        /* Increment ”j” by ”j0”. If j equals n0, then increment both ”x0” */
        /* and ”x2” so that double inversion is avoided.                   */
        /*-----------------------------------------------------------------*/
        j += j0;
        if (j == n0)
        {
           j  += n0;
           x0 += (int) npoints>>1;
           x2 += (int) npoints>>1;
        }
    }
}
```

Special Requirements

- In-place computation is *not* allowed.
- The size of the FFT, nx, must be power of 2 or 4, and 16 ≤ nx ≤ 32768.
- The arrays for the complex input data x[ ], complex output data y[ ], and twiddle factors w[ ] must be double-word aligned.
- The input and output data are complex, with the real/imaginary components stored in adjacent locations in the array. The real components are stored at even array indices, and the imaginary components are stored at odd array indices. All data are in short precision or Q.15 format.
- The FFT coefficients (twiddle factors) are generated using the program tw_fft16x16 provided in the directory ‘support\fft’. The scale factor must be 32767.5. No scaling is done with the function; thus, the input data must be scaled by $2^{\log2(nx)}$ to completely prevent overflow.

Implementation Notes

- ***Bank Conflicts:*** nx/8 bank conflicts occur.
- ***Interruptibility:*** The code is interrupt-tolerant but not interruptible.

The routine uses $\log_4(nx) - 1$ stages of radix-4 transform and performs either a radix-2 or radix-4 transform on the last stage depending on nx. If nx is a power of 4, then this last stage is also a radix-4 transform, otherwise it is a radix-2 transform. The conventional Cooley Tukey FFT is written using three loops. The outermost loop “k” cycles through the stages. There are log N to the base 4 stages in all. The loop “j” cycles through the groups of butterflies with different twiddle factors, and loop “i” reuses the twiddle factors for the different butterflies within a stage. Note the following:

Stage	Groups	Butterflies With Common Twiddle Factors	Groups*Butterflies
1	N/4	1	N/4
2	N/16	4	N/4
..	..	..	..
logN	1	N/4	N/4

The following statements can be made based on above observations:

1) Inner loop “i0” iterates a variable number of times. In particular, the number of iterations quadruples every time from 1..N/4. Hence, software pipelining a loop that iterates a variable number of times is not profitable.

2) Outer loop “j” iterates a variable number of times as well. However, the number of iterations is quartered every time from N/4 ..1. Hence, the behavior in (a) and (b) are exactly opposite to each other.

3) If the two loops “i” and “j” are coalesced together, then they will iterate for a fixed number of times, namely N/4. This allows us to combine the “i” and “j” loops into one loop. Optimized implementations will make use of this fact.

In addition, the Cooley Tukey FFT accesses three twiddle factors per iteration of the inner loop, as the butterflies that re-use twiddle factors are lumped together. This leads to accessing the twiddle factor array at three points each separated by “ie”. Note that “ie” is initially 1, and is quadrupled with every iteration. Therefore, these three twiddle factors are not even contiguous in the array.

To vectorize the FFT, it is desirable to access twiddle factor array using double word wide loads and fetch the twiddle factors needed. To do this, a modified twiddle factor array is created, in which the factors WN/4, WN/2, W3N/4 are arranged to be contiguous. This eliminates the separation between twiddle factors within a butterfly. However, this implies that we maintain a redundant version of the twiddle factor array as the loop is traversed from one stage to another. Hence, the size of the twiddle factor array increases as compared to the normal Cooley Tukey FFT. The modified twiddle factor array is of size “2 * N”, where the conventional Cooley Tukey FFT is of size “3N/4”, where N is the number of complex points to be transformed. The routine that generates the modified twiddle factor array was presented earlier. With the above transformation of the FFT, both the input data and the twiddle factor array can be accessed using double-word wide loads to enable packed data processing.

The final stage is optimized to remove the multiplication as w0 = 1. This stage also performs digit reversal on the data, so the final output is in natural order. In addition, if the number of points to be transformed is a power of 2, the final stage applies a DSP_radix2 pass instead of a radix 4. In any case, the outputs are returned in normal order.

The code shown here performs the bulk of the computation in place. However, because digit-reversal cannot be performed in-place, the final result is written to a separate array, y[].

There is one slight break in the flow of packed processing. The real part of the complex number is in the lower half, and the imaginary part is in the upper half. The flow breaks for “xl0” and “xl1” because in this case the real part needs to be combined with the imaginary part because of the multiplication by “j”. This requires a packed quantity like “xl21xl20” to be rotated as “xl20xl21” so that it can be combined using ADD2s and SUB2s. Hence, the natural version of C code shown below is transformed using packed data processing as shown:

```
xl0  = x[2 * i0    ] − x[2 * i2    ];
xl1  = x[2 * i0 + 1] − x[2 * i2 + 1];
xl20 = x[2 * i1    ] − x[2 * i3    ];
xl21 = x[2 * i1 + 1] − x[2 * i3 + 1];

xt1  = xl0 + xl21;
yt2  = xl1 + xl20;
xt2  = xl0 − xl21;
yt1  = xl1 − xl20;

xl1_xl0   = _sub2(x21_x20, x21_x20)
xl21_xl20 = _sub2(x32_x22, x23_x22)
xl20_xl21 = _rotl(xl21_xl20, 16)

yt2_xt1   = _add2(xl1_xl0, xl20_xl21)
yt1_xt2   = _sub2(xl1_xl0, xl20_xl21)
```

Also notice that xt1, yt1 end up on separate words, these need to be packed together to take advantage of the packed twiddle factors that have been loaded. To achiev this, they are re-aligned as follows:

```
yt1_xt1 = _packhl2(yt1_xt2, yt2_xt1)
yt2_xt2 = _packhl2(yt2_xt1, yt1_xt2)
```

The packed words “yt1_xt1” allow the loaded “sc” twiddle factor to be used for the complex multiplies. The real part of the complex multiply is implemented using DOTP2. The imaginary part of the complex multiply is implemented using DOTPN2 after the twiddle factors are swizzled within the half word.

$(X + jY)\ (C + j\ S) = (XC + YS) + j\ (YC - XS)$.

The actual twiddle factors for the FFT are cosine, − sine. The twiddle factors stored in the table are cosine and sine, hence the sign of the ”sine” term is comprehended during multiplication as shown above.

Benchmarks

Cycles	$(10 * nx/8 + 19) * \text{ceil}[\log_4(nx) - 1] + (nx/8 + 2) * 7 + 28 + BC$ where BC = N/8, the number of bank conflicts.
Codesize	1004 bytes

Appendix A

Performance/Fractional Q Formats

This appendix describes performance considerations related to the C64x+ DSPLIB and provides information about the Q format used by DSPLIB functions.

Topic **Page**

A.1 Performance Considerations

The ceil( ) is used in some benchmark formulas to accurately describe the number of cycles. It returns a number rounded up, away from zero, to the nearest integer. For example, ceil(1.1) returns 2.

Although DSPLIB can be used as a first estimation of processor performance for a specific function, you should be aware that the generic nature of DSPLIB might add extra cycles not required for customer specific usage.

Benchmark cycles presented assume best case conditions, typically assuming all code and data are placed in L1 memory. Any extra cycles due to placement of code or data in L2/external memory or cache-associated effects (cache-hits or misses) are not considered when computing the cycle counts.

You should also be aware that execution speed in a system is dependent on where the different sections of program and data are located in memory. You should account for such differences when trying to explain why a routine is taking more time than the reported DSPLIB benchmarks.

For more information on additional stall cycles due to memory hierarchy, see the *Signal Processing Examples Using TMS320C64x Digital Signal Processing Library* (SPRA884). The *TMS320C6000 DSP Cache User’s Guide* (SPRU656A) presents how to optimize algorithms and function calls for better cache performance.

A.2 Fractional Q Formats

Unless specifically noted, DSPLIB functions use Q15 format, or to be more exact, Q0.15. In a Q*m*.*n* format, there are *m* bits used to represent the two's complement integer portion of the number, and *n* bits used to represent the two's complement fractional portion. *m+n+1* bits are needed to store a general Q*m*.*n* number. The extra bit is needed to store the sign of the number in the most-significant bit position. The representable integer range is specified by $(-2^m, 2^m)$ and the finest fractional resolution is 2^{-n}.

For example, the most commonly used format is Q.15. Q.15 means that a 16-bit word is used to express a signed number between positive and negative one. The most-significant binary digit is interpreted as the sign bit in any Q format number. Thus, in Q.15 format, the decimal point is placed immediately to the right of the sign bit. The fractional portion to the right of the sign bit is stored in regular two's complement format.

A.2.1 Q3.12 Format

Q.3.12 format places the sign bit after the fourth binary digit from the right, and the next 12 bits contain the two's complement fractional component. The approximate allowable range of numbers in Q.3.12 representation is (−8,8) and the finest fractional resolution is $2^{-12} = 2.441 \times 10^{-4}$.

Table A−1. Q3.12 Bit Fields

Bit	15	14	13	12	11	10	9	…	0
Value	S	I3	I2	I1	Q11	Q10	Q9	…	Q0

A.2.2 Q.15 Format

Q.15 format places the sign bit at the leftmost binary digit, and the next 15 leftmost bits contain the two's complement fractional component. The approximate allowable range of numbers in Q.15 representation is (−1,1) and the finest fractional resolution is $2^{-15} = 3.05 \times 10^{-5}$.

Table A−2. Q.15 Bit Fields

Bit	15	14	13	12	11	10	9	…	0
Value	S	Q14	Q13	Q12	Q11	Q10	Q9	…	Q0

A.2.3 Q.31 Format

Q.31 format spans two 16-bit memory words. The 16-bit word stored in the lower memory location contains the 16 least significant bits, and the higher memory location contains the most significant 15 bits and the sign bit. The approximate allowable range of numbers in Q.31 representation is (−1,1) and the finest fractional resolution is $2^{-31} = 4.66 \times 10^{-10}$.

Table A−3. Q.31 Low Memory Location Bit Fields

Bit	15	14	13	12	…	3	2	1	0
Value	Q15	Q14	Q13	Q12	…	Q3	Q2	Q1	Q0

Table A−4. Q.31 High Memory Location Bit Fields

Bit	15	14	13	12	…	3	2	1	0
Value	S	Q30	Q29	Q28	…	Q19	Q18	Q17	Q16

Appendix B

Software Updates and Customer Support

This appendix provides information about software updates and customer support.

Topic **Page**

B.1 DSPLIB Software Updates

C64x DSPLIB software updates may be periodically released incorporating product enhancements and fixes as they become available. You should read the README.TXT available in the root directory of every release.

B.2 DSPLIB Customer Support

If you have questions or want to report problems or suggestions regarding the C64x DSPLIB, contact Texas Instruments at dsph@ti.com.

Appendix C

Glossary

A

address: The location of program code or data stored; an individually accessible memory location.

A-law companding: See *compress and expand (compand)*.

API: See *application programming interface.*

application programming interface (API): Used for proprietary application programs to interact with communications software or to conform to protocols from another vendor's product.

assembler: A software program that creates a machine language program from a source file that contains assembly language instructions, directives, and macros. The assembler substitutes absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

assert: To make a digital logic device pin active. If the pin is active low, then a low voltage on the pin asserts it. If the pin is active high, then a high voltage asserts it.

B

bit: A binary digit, either a 0 or 1.

big endian: An addressing protocol in which bytes are numbered from left to right within a word. More significant bytes in a word have lower numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *little endian*.

block: The three least significant bits of the program address. These correspond to the address within a fetch packet of the first instruction being addressed.

board support library (BSL): The BSL is a set of application programming interfaces (APIs) consisting of target side DSP code used to configure and control board level peripherals.

boot: The process of loading a program into program memory.

boot mode: The method of loading a program into program memory. The C6x DSP supports booting from external ROM or the host port interface (HPI).

BSL: See *board support library.*

byte: A sequence of eight adjacent bits operated upon as a unit.

C

cache: A fast storage buffer in the central processing unit of a computer.

cache controller: System component that coordinates program accesses between CPU program fetch mechanism, cache, and external memory.

CCS: Code Composer Studio.

central processing unit (CPU): The portion of the processor involved in arithmetic, shifting, and Boolean logic operations, as well as the generation of data- and program-memory addresses. The CPU includes the central arithmetic logic unit (CALU), the multiplier, and the auxiliary register arithmetic unit (ARAU).

chip support library (CSL): The CSL is a set of application programming interfaces (APIs) consisting of target side DSP code used to configure and control all on-chip peripherals.

clock cycle: A periodic or sequence of events based on the input from the external clock.

clock modes: Options used by the clock generator to change the internal CPU clock frequency to a fraction or multiple of the frequency of the input clock signal.

code: A set of instructions written to perform a task; a computer program or part of a program.

coder-decoder or compression/decompression (codec): A device that codes in one direction of transmission and decodes in another direction of transmission.

compiler: A computer program that translates programs in a high-level language into their assembly-language equivalents.

compress and expand (compand): A quantization scheme for audio signals in which the input signal is compressed and then, after processing, is reconstructed at the output by expansion. There are two distinct companding schemes: A-law (used in Europe) and μ-law (used in the United States).

control register: A register that contains bit fields that define the way a device operates.

control register file: A set of control registers.

CSL: See *chip support library*.

D

device ID: Configuration register that identifies each peripheral component interconnect (PCI).

digital signal processor (DSP): A semiconductor that turns analog signals—such as sound or light—into digital signals, which are discrete or discontinuous electrical impulses, so that they can be manipulated.

direct memory access (DMA): A mechanism whereby a device other than the host processor contends for and receives mastery of the memory bus so that data transfers can take place independent of the host.

DMA : See *direct memory access*.

DMA source: The module where the DMA data originates. DMA data is read from the DMA source.

DMA transfer: The process of transferring data from one part of memory to another. Each DMA transfer consists of a read bus cycle (source to DMA holding register) and a write bus cycle (DMA holding register to destination).

DSP_autocor: Autocorrelation.

DSP_bexp: Block exponent implementation.

DSP_bitrev_cplx: Complex bit reverse.

DSP_blk_eswap16: Endian-swap a block of 16-bit values.

DSP_blk_eswap32: Endian-swap a block of 32-bit values.

DSP_blk_eswap64: Endian-swap a block of 64-bit values.

DSP_blk_move: Block move.

DSP_dotp_sqr: Vector dot product and square.

DSP_dotprod: Vector dot product.

DSP_fft: Complex forward FFT with digital reversal.

DSP_fft16x16r: Complex forward mixed radix 16- x 16-bit FFT with rounding.

DSP_fft16x16t: Complex forward mixed radix 16- x 16-bit FFT with truncation.

DSP_fft16x32: Complex forward mixed radix 16- x 32-bit FFT with rounding.

DSP_fft32x32: Complex forward mixed radix 32- x 32-bit FFT with rounding.

DSP_fft32x32s: Complex forward mixed radix 32- x 32-bit FFT with scaling.

DSP_fir_cplx: Complex FIR filter (radix 2).

DSP_fir_gen: FIR filter (general purpose).

DSP_firlms2: LMS FIR (radix 2).

DSP_fir_r4: FIR filter (radix 4).

DSP_fir_r8: FIR filter (radix 8).

DSP_fir_sym: Symmetric FIR filter (radix 8).

DSP_fltoq15: Float to Q15 conversion.

DSP_ifft16x32: Complex inverse mixed radix 16- x 32-bit FFT with rounding.

DSP_ifft32x32: Complex inverse mixed radix 32- x 32-bit FFT with rounding.

DSP_iir: IIR with 5 coefficients per biquad.

DSP_mat_mul: Matrix multiplication.

DSP_mat_trans: Matrix transpose.

DSP_maxidx: Index of the maximum element of a vector.

DSP_maxval: Maximum value of a vector.

DSP_minerror: Minimum energy error search.

DSP_minval: Minimum value of a vector.

DSP_mul32: 32-bit vector multiply.

DSP_neg32: 32-bit vector negate.

DSP_q15tofl: Q15 to float conversion.

DSP_radix2: Complex forward FFT (radix 2).

DSP_recip16: 16-bit reciprocal.

DSP_r4fft: Complex forward FFT (radix 4).

DSP_vecsumsq: Sum of squares.

DSP_w_vec: Weighted vector sum.

E

evaluation module (EVM): Board and software tools that allow the user to evaluate a specific device.

external interrupt: A hardware interrupt triggered by a specific value on a pin.

external memory interface (EMIF): Microprocessor hardware that is used to read to and write from off-chip memory.

F

fast Fourier transform (FFT): An efficient method of computing the discrete Fourier transform algorithm, which transforms functions between the time domain and the frequency domain.

fetch packet: A contiguous 8-word series of instructions fetched by the CPU and aligned on an 8-word boundary.

FFT: See *fast fourier transform.*

flag: A binary status indicator whose state indicates whether a particular condition has occurred or is in effect.

frame: An 8-word space in the cache RAMs. Each fetch packet in the cache resides in only one frame. A cache update loads a frame with the requested fetch packet. The cache contains 512 frames.

G

global interrupt enable bit (GIE): A bit in the control status register (CSR) that is used to enable or disable maskable interrupts.

H

HAL: *Hardware abstraction layer* of the CSL. The HAL underlies the service layer and provides it a set of macros and constants for manipulating the peripheral registers at the lowest level. It is a low-level symbolic interface into the hardware providing symbols that describe peripheral registers/bitfields, and macros for manipulating them.

host: A device to which other devices (peripherals) are connected and that generally controls those devices.

host port interface (HPI): A parallel interface that the CPU uses to communicate with a host processor.

HPI: See *host port interface*; see also *HPI module*.

I

index: A relative offset in the program address that specifies which frame is used out of the 512 frames in the cache into which the current access is mapped.

indirect addressing: An addressing mode in which an address points to another pointer rather than to the actual data; this mode is prohibited in RISC architecture.

instruction fetch packet: A group of up to eight instructions held in memory for execution by the CPU.

internal interrupt: A hardware interrupt caused by an on-chip peripheral.

interrupt: A signal sent by hardware or software to a processor requesting attention. An interrupt tells the processor to suspend its current operation, save the current task status, and perform a particular set of instructions. Interrupts communicate with the operating system and prioritize tasks to be performed.

interrupt service fetch packet (ISFP): A fetch packet used to service interrupts. If eight instructions are insufficient, you must branch out of this block for additional interrupt service. If the delay slots of the branch do not reside within the ISFP, execution continues from execute packets in the next fetch packet (the next ISFP).

interrupt service routine (ISR): A module of code that is executed in response to a hardware or software interrupt.

interrupt service table (IST) A table containing a corresponding entry for each of the 16 physical interrupts. Each entry is a single-fetch packet and has a label associated with it.

Internal peripherals: Devices connected to and controlled by a host device. The C6x internal peripherals include the direct memory access (DMA) controller, multichannel buffered serial ports (McBSPs), host port interface (HPI), external memory-interface (EMIF), and runtime support timers.

IST: See *interrupt service table.*

L

least significant bit (LSB): The lowest-order bit in a word.

linker: A software tool that combines object files to form an object module, which can be loaded into memory and executed.

little endian: An addressing protocol in which bytes are numbered from right to left within a word. More significant bytes in a word have higher-numbered addresses. Endian ordering is specific to hardware and is determined at reset. See also *big endian*.

M

maskable interrupt: A hardware interrupt that can be enabled or disabled through software.

memory map: A graphical representation of a computer system's memory, showing the locations of program space, data space, reserved space, and other memory-resident elements.

memory-mapped register: An on-chip register mapped to an address in memory. Some memory-mapped registers are mapped to data memory, and some are mapped to input/output memory.

most significant bit (MSB): The highest order bit in a word.

μ-law companding: See *compress and expand (compand)*.

multichannel buffered serial port (McBSP): An on-chip full-duplex circuit that provides direct serial communication through several channels to external serial devices.

multiplexer: A device for selecting one of several available signals.

N

nonmaskable interrupt (NMI): An interrupt that can be neither masked nor disabled.

O

object file: A file that has been assembled or linked and contains machine language object code.

off chip: A state of being external to a device.

on chip: A state of being internal to a device.

P

peripheral: A device connected to and usually controlled by a host device.

program cache: A fast memory cache for storing program instructions allowing for quick execution.

program memory: Memory accessed through the C6x’s program fetch interface.

PWR: Power; see *PWR module*.

PWR module: PWR is an API module that is used to configure the power-down control registers, if applicable, and to invoke various power-down modes.

R

random-access memory (RAM): A type of memory device in which the individual locations can be accessed in any order.

register: A small area of high speed memory located within a processor or electronic device that is used for temporarily storing data or instructions. Each register is given a name, contains a few bytes of information, and is referenced by programs.

reduced-instruction-set computer (RISC): A computer whose instruction set and related decode mechanism are much simpler than those of microprogrammed complex instruction set computers. The result is a higher instruction throughput and a faster real-time interrupt service response from a smaller, cost-effective chip.

reset: A means of bringing the CPU to a known state by setting the registers and control bits to predetermined values and signaling execution to start at a specified address.

RTOS *Real-time operating system.*

S

service layer: The top layer of the 2-layer chip support library architecture providing high-level APIs into the CSL and BSL. The service layer is where the actual APIs are defined and is the interface layer.

synchronous-burst static random-access memory (SBSRAM): RAM whose contents do not have to be refreshed periodically. Transfer of data is at a fixed rate relative to the clock speed of the device, but the speed is increased.

synchronous dynamic random-access memory (SDRAM): RAM whose contents are refreshed periodically so the data is not lost. Transfer of data is at a fixed rate relative to the clock speed of the device.

syntax: The grammatical and structural rules of a language. All higher-level programming languages possess a formal syntax.

system software: The blanketing term used to denote collectively the chip support libraries and board support libraries.

T

tag: The 18 most significant bits of the program address. This value corresponds to the physical address of the fetch packet that is in that frame.

timer: A programmable peripheral used to generate pulses or to time events.

TIMER module: TIMER is an API module used for configuring the timer registers.

W

word: A multiple of eight bits that is operated upon as a unit. For the C6x, a word is 32 bits in length.

Index